Tutorial

django girls

Tabla de contenido

Introducción	1.1
Instalación	1.2
Instalación (chromebook)	1.3
Cómo funciona Internet	1.4
Introducción a la línea de comandos	1.5
Instalación de Python	1.6
Editor de código	1.7
Introducción a Python	1.8
¿Qué es Django?	1.9
Instalación de Django	1.10
¡Tu primer proyecto en Django!	1.11
Modelos en Django	1.12
Administrador de Django	1.13
¡Despliega!	1.14
URLs en Django	1.15
Vistas en Django - ¡Hora de crear!	1.16
Introducción a HTML	1.17
ORM de Django (QuerySets)	1.18
Datos dinámicos en las plantillas	1.19
Plantillas de Django	1.20
CSS - ¡Que quede bonito!	1.21
Extendiendo plantillas	1.22
Amplía tu aplicación	1.23
Formularios de Django	1.24
¿Y ahora qué?	1.25

Tutorial de Django Girls

chat on gitter

> Este trabajo está bajo la licencia internacional Creative Commons Attribution-ShareAlike 4.0. Para ver una copia de esta licencia, visita el siguiente enlace https://creativecommons.org/licenses/by-sa/4.0/

Bienvenido/a

¡Bienvenido/a al tutorial de las Django Girls! ¡Nos alegra que estes aquí :) En este tutorial, te llevaremos de viaje a las entrañas de la tecnología web, para que veas todas las piezas que se necesitan para que la web funcione.

Como pasa con todas las cosas nuevas, va a ser una aventura - pero no te preocupes; una vez que te has decidido a empezar, te va a ir fenomenal :)

Introducción

¿Alguna vez has tenido la sensación de que el mundo es cada vez más tecnológico? ¿que cada vez lo entiendes menos? ¿Alguna vez te has planteado crear un sitio web pero no sabías por donde empezar? ¿Has pensado alguna vez que el mundo del software es demasiado complicado como para intentar hacer algo por tu cuenta?

Bueno, ¡tenemos buenas noticias! Programar no es tan difícil como parece y queremos demostrarte lo divertido puede llegar a ser.

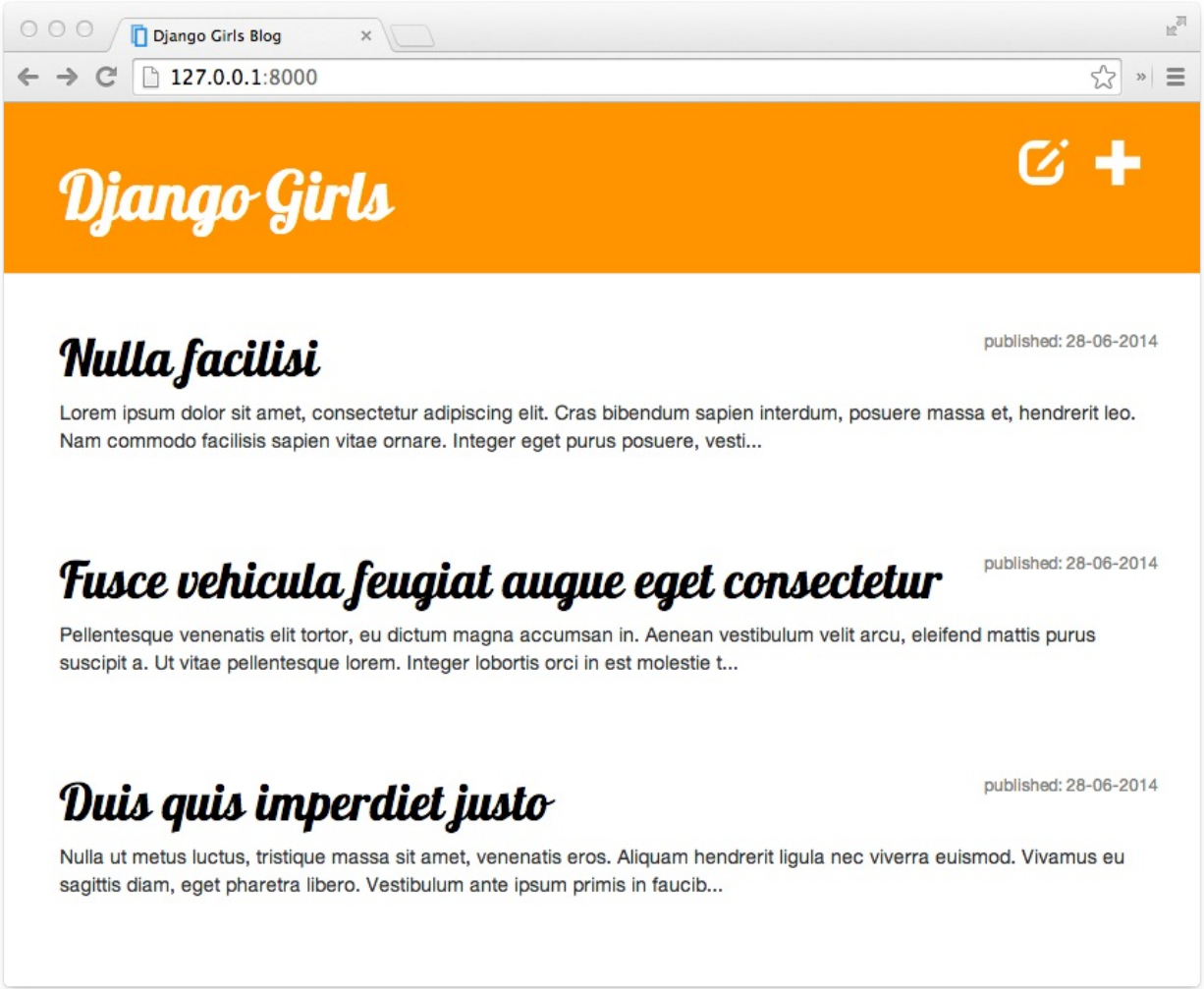
Este tutorial no te convertirá en programadora de la noche a la mañana. Sí quieres ser buena en esto, necesitarás meses o incluso años de aprendizaje y práctica. Sin embargo queremos enseñarte que programar o crear sitios web no es tan complicado como parece. Intentaremos explicar las cosas lo mejor que podamos, para perderle el miedo a la tecnología.

¡Esperamos conseguir que te guste la tecnología tanto como a nosotras!

¿Qué aprenderás con este tutorial?

Cuando termines el tutorial, tendrás una aplicación web sencilla y funcional: tu propio blog. Te mostraremos como ponerla en línea, ¡para que otros puedan ver tu trabajo!

Tendrá (más o menos) esta pinta:



> Si estás siguiendo este tutorial por tu cuenta y no tienes un mentor que te ayude en caso de dificultades, tenemos un chat para ti: chat on gitter. ¡Hemos pedido a mentores y asistentes de ediciones anteriores, que pasen por allí de vez en cuando para echar una mano a otras con el tutorial! ¡No tengas miedo de preguntar ahí!

Bien, empecemos por el comienzo...

Seguir el tutorial desde casa

Participar en un taller de Django Girls en vivo es genial, pero somos conscientes de que no siempre es posible hacerlo. Por eso, te recomendamos hacer este tutorial en casa. Para las que estáis casa, estamos preparando videos que facilitarán seguir el tutorial por tu cuenta. Todavía está en progreso, pero cada vez hay más cosas explicadas en el canal de YouTube Coding is for girls (Nota: en inglés).

En cada capítulo hay un enlace que lleva al video correspondiente (si lo hay).

Sobre nosotras y Cómo contribuir

Este tutorial lo mantiene DjangoGirls. Si encuentras algún error o quieres actualizar el tutorial, por favor sigue la guía de cómo contribuir.

¿Te gustaría ayudarnos a traducir el tutorial a otros idiomas?

Actualmente, las traducciones se hacen en la plataforma crowdin.com, en el siguiente enlace:

https://crowdin.com/project/django-girls-tutorial

Si tu idioma no aparece en la lista de crowdin, por favor abre un nuevo issue con el idioma para que podamos añadirlo.

Si estás haciendo el tutorial en casa

Si estás haciendo el tutorial en casa, y no en uno de los eventos de Django Girls, puedes saltar este capítulo por completo e ir directamente al capítulo ¿cómo funciona Internet?.

Esto es porque cubrimos estas cosas en todo el tutorial de todos modos, y esto es sólo una página adicional que recoge todas las instrucciones de instalación en un solo lugar. El evento de Django Grils incluye una "noche de la instalación" en la que instalamos todo, así que no tenemos que molestar con ella durante el taller, así que esto es útil para nosotros.

Si lo encuentras útil, puedes seguir este capítulo también. Pero si quieres empezar a aprender antes de instalar un montón de cosas en tu computadora, sáltate este capítulo y te explicaremos la parte de instalación más adelante.

¡Buena suerte!

Instalación

En el taller construiremos un blog, y hay algunas tareas de configuración en el tutorial que sería bueno hacer de antemano para que estés listo para comenzar a codificar en el día.

Chromebook setup (if you're using one) Puedes [saltarte esta sección](http://tutorial.djangogirls.org/en/installation/#install-python) en caso de que no estés usando un Chromebook. Si lo estás usando, tu experiencia de instalación será algo diferente. Puedes ignorar el resto de las instrucciones de instalación. ### IDE en la nube (PaizaCloud Cloud IDE, AWS Cloud9) Un IDE en la nube es una herramienta que te da un editor de código y acceso a un ordenador conectado a internet en el que puedes instalar, escribir y ejecutar software. En este tutorial, el IDE en la nube te servirá como tu *máquina local*. Seguirás ejecutando comandos en una terminal igual que tus compañeros de clase en OS X, Ubuntu, o Windows, pero tu terminal en realidad estará conectada a un ordenador corriendo en otro sitio que el IDE en la nube gestionará para ti. Aquí están las instrucciones para IDEs en la nube (PaizaCloud Cloud IDE, AWS Cloud9). Puedes elegir uno de los IDEs en la nube, y seguir sus instrucciones. #### PaizaCloud Cloud IDE 1. Ve a [PaizaCloud Cloud IDE](https://paiza.cloud/) 2. Crea una cuenta 3. Haz click en *New Server* 4. Haz click en el botón Terminal (en el lado izquierdo de la ventana) Ahora deberías ver una interfaz con una barra y botones en la izquierda. Haz click en al botón "Terminal" para abrir la ventana de la terminal con un símbolo de sistema como este:
Terminal

$ La terminal enviará tus instrucciones al ordenador que Cloud 9 ha preparado para ti. Puedes redimensionar o maximizar la ventana para hacerla un poco mas grande. #### AWS Cloud9 1. Ve a [AWS Cloud9](https://aws.amazon.com/cloud9/) 2. Crea una cuenta 3. Haz click en *Create Environment* Deberías ver un interfaz con una barra lateral, una ventana principal grande con texto y una ventana pequeña abajo del todo parecida a esto:
bash

yourusername:~/workspace $ La parte inferior es tu *terminal*, donde escribirás las instrucciones para el ordenador que Cloud 9 te ha preparado. Puedes redimensionar la ventana para hacerla un poco más grande. ### Entorno Virtual Un entorno virtual (también llamado virtualenv) es como una caja privada donde podemos guardar código útil para el proyecto en el que estamos trabajando. Lo usamos para guardar por separado los trozos de código de distintos proyectos, y que así, las cosas no se mezclen entre proyectos. En la terminal de la parte inferior de Cloud 9, ejecuta lo siguiente:
Cloud 9

sudo apt update sudo apt install python3.6-venv Si aun así, no te funciona, pide ayuda a tu mentor. A continuación, ejecuta:
Cloud 9

mkdir djangogirls cd djangogirls python3.6 -mvenv myvenv source myvenv/bin/activate pip install django~=2.0.6 (fíjate que en la última línea hemos usado una tilde seguida de un signo de igual: ~=). ### GitHub Hazte una cuenta de [GitHub](https://github.com). ### Python Anywhere El tutorial de Django Girls tiene una sección que se llama "Despliegue", que es

el proceso de coger el código de tu nueva aplicación web y ponerlo en un ordenador públicamente accesible (un servidor) para que todo el mundo pueda ver tu trabajo. Esta parte "chirría" un poco cuando haces el tutorial en un Chromebook, porque que ya estamos usando un ordenador que ya está en Internet (en lugar de un ordenador local como un portátil). Sin embargo, aún tiene sentido, si pensamos que nuestro espacio de trabajo en Cloud9 es un lugar para nuestro trabajo "en progreso" y PythonAnywhere es el lugar donde enseñar nuestro sitio web más terminado. Así que créate una cuenta de PythonAnywhere en www.pythonanywhere.com.

Instalar Python

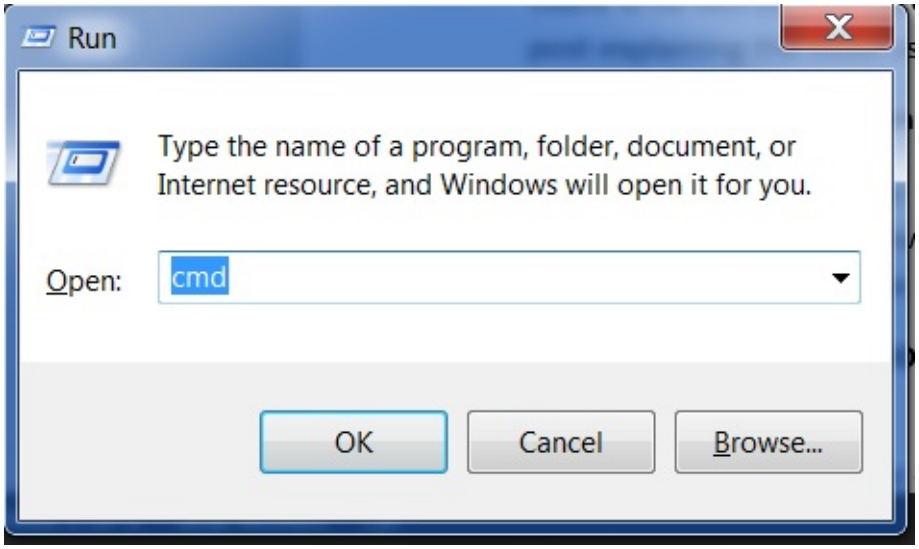
> Para lectores en casa: este capitulo se cubre en el video Installing Python & Code Editor.
>
> Esta sección está basada en un tutorial de Geek Girls Carrots (https://github.com/ggcarrots/django-carrots)

Django esta escrito en Python. Necesitamos Python para hacer cualquier cosa en Django. Comencemos instalandolo!. Tenemos que instalar Python 3.6, así que si tienes una versión anterior, debes actualizarla.

Install Python: Windows
Primero comprueba si tu computador tiene una version de Windows de 32-bit o 64-bit, presiona tecla Windows + Pause/Break, esto abrirá tu System info (información de tu sistema), ahora busta la linea "System type". Puedes descargar Python para Windows desde el sitio web https://www.python.org/downloads/windows/. Haz click en el link "Latest Python 3 release - Python x.x.x.". Si tu computador tiene una versión de Windows de **64 bits**, descarga **Windows x86-64 executable installer**. De lo contrario, descarga **Windows x86 executable installer**. Después de descargar el instalador, debes ejecutarlo (hazle doble click) y sigue las instrucciones.

Algo a tener en cuenta: Durante la instalación notarás una ventana llamada "Setup". Asegúrate de seleccionar la casilla "Add Python 3.6 to Path" y luego haz click en "Install Now", como se muestra a continuación:

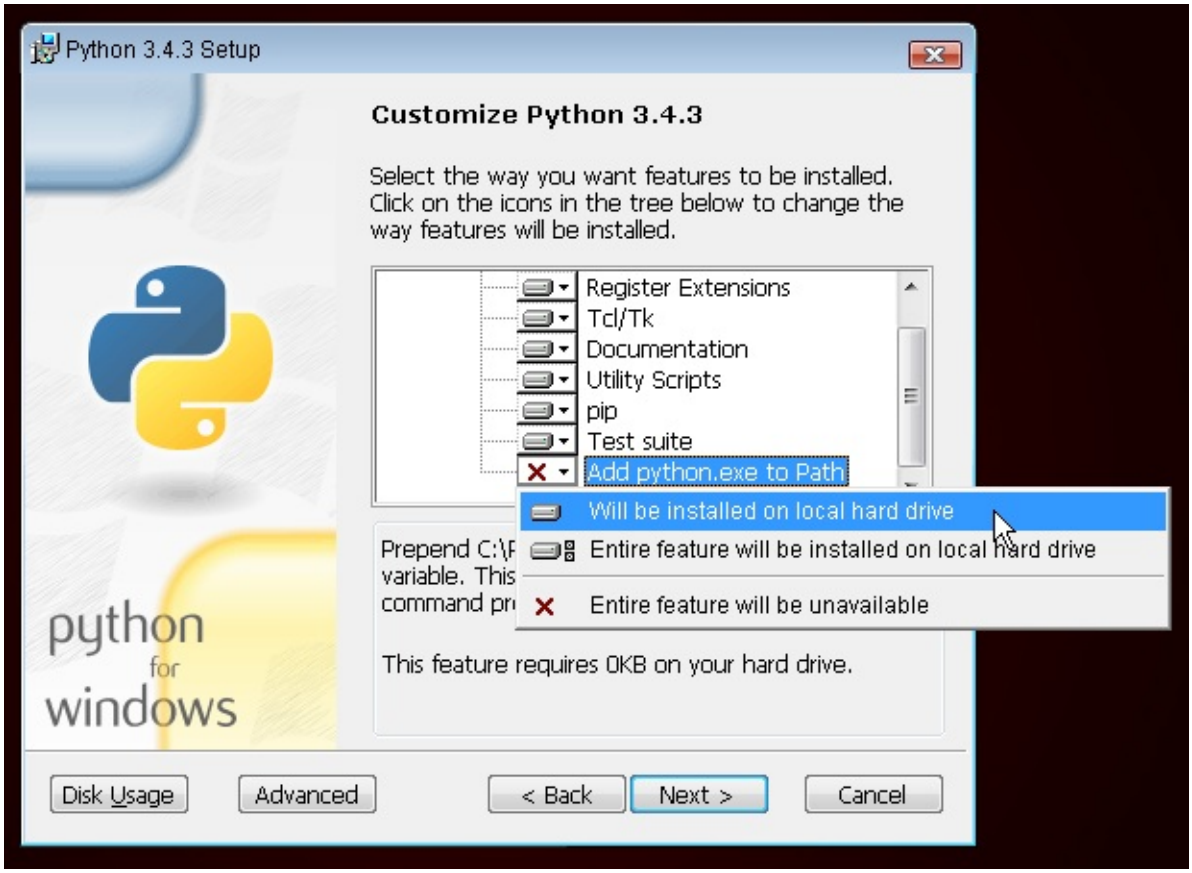
En los próximos pasos, vas a utilizar la línea de comandos de Windows (acerca de la cual también de contaremos algo). De momento, si tienes que teclear algunos comandos, ve al menú de Inicio y teclea "Command Prompt" en el cuadro de búsqueda. (En versiones anteriores de Windows, puedes arrancar la linea de comandos con menú de Inicio → Sistema Windows → Linea de Comandos.) También puedes pulsar la tecla "Windows" + R hasta que aparezca la ventana "Ejecutar" (Run). Para abrir la línea de comandos, escribe "cmd" y pulsa enter en la ventana "Run".



Nota: Si estas utilizando una versión anterior de Windows (7, Vista, o cualquier versión anterior) y el instalador de Python 3.6.x falla con un error, puedes tratar ya sea:

1. instalar todas las actualizaciones de Windows e intentar instalar Python 3.6 nuevamente; o
2. instalar una versión de Python anterior, por ejemplo, 3.4.6.

Si instalas una versión anterior de Python, la pantalla de instalación puede ser un poco diferente a la que se muestra arriba. Asegúrate de desplazarte hacia abajo para ver "Add python.exe to Path", y dar click en el botón a la izquierda y seleccionar "Will be installed on local hard drive":



Install Python: OS X

> **Nota** Antes de instalar Python en OS X, debes asegurarte de que la configuración del Mac permita instalar paquetes que no estén en la App Store. ve a preferencias del sistema (System Preferences, está en la carpeta Aplicaciones), da click en "Seguridad y privacidad" (Security & Privacy) y luego la pestaña "General". Si tu "Permitir aplicaciones descargadas desde:" (Allow apps downloaded from:) está establecida a "Mac App Store," cambia a "Mac App Store y desarrolladores identificados." (Mac App Store and identified developers)

Debes ir al sitio web https://www.python.org/downloads/release/python-361/ y descargar el instalador de Python:

- Descargar el archivo *Mac OS X 64-bit/32-bit installer*,
- Haz doble clic en *python-3.4.3-macosx10.6.pkg* para ejecutar al instalador.

Install Python: Linux
Es muy posible que ya tengas Python instalado de serie. Para verificar que ya lo tienes instalado (y qué versión es), abre una consola y escribe el siguiente comando:

command-line

```
$ python3 --version
Python 3.6.1
```

Si tienes una 'micro versión' diferente de Python instalada, por ejemplo 3.6.0, entonces no tienes que actualizar. Si no tienes instalado Python o si deseas una versión diferente, puedes instalarla de la siguiente manera:

Install Python: Debian or Ubuntu
Escribe este comando en tu consola:

command-line

```
$ sudo apt install python3.6
```

Install Python: Fedora
Usa este comando en tu consola:

command-line

```
$ sudo dnf instalar python3
```

En versiones anteriores de Fedora tal vez te salga un error de que no se encuentra el comando `dnf`. En ese caso utiliza `yum` en su lugar.

Install Python: openSUSE
Usa este comando en tu consola:

command-line

```
$ sudo zypper install python3
```

Verifica que la instalación fue exitosa abriendo una terminal y ejecutando el comando `python3`:

command-line

```
$ python3 --version
Python 3.6.1
```

NOTA: Si estás en Windows y recibes un mensaje de error no se encontró `python3`, intenta usar `python` (sin el `3`) y comprueba si todavía es una versión de Python 3.6.

Si tienes alguna duda o si algo salió mal y no sabes cómo resolverlo - ¡pide ayuda a tu tutor! A veces las cosas no van bien y que es mejor pedir ayuda a alguien con más experiencia.

Instala un Editor de Código

Hay muchos editores diferentes y la elección es principalmente una cuestión de preferencia personal. La mayoría de programadores de Python usan IDEs (Entornos de Desarrollo Integrados) complejos pero muy potentes, como PyCharm. Sin embargo, como principiante, probablemente no es lo más aconsejable; nuestras recomendaciones son igualmente potentes pero mucho más simples.

Abajo presentamos nuestras sugerencias pero, también puedes preguntarle a tu mentor cuáles son las suyas - será más fácil que te ayude.

Gedit

Gedit es un editor de texto de código abierto, gratuito, y que está disponible para todos los sistemas operativos.

Descárgalo aquí

Sublime Text 3

Sublime Text es un editor muy popular con un período de prueba gratis. Es fácil de instalar y de usar, y está disponible para todos los sistemas operativos.

Descárgalo aquí

Atom

Atom es otro editor que usa mucha gente. Es gratuito, de código abierto y hay versiones para Windows, OS X y Linux. Atom ha sido desarrollado por GitHub.

Descárgalo aquí

¿Por qué estamos instalando un editor de texto?

Tal vez te estés preguntando por qué estamos instalando un editor especializado en código, en lugar de usar un editor convencional como Word o Notepad.

La principal razón es que el código tiene que ser **texto plano** y, el problema con programas como Word y Textedit es que no guardan texto plano, sino que guardan texto enriquecido (con estilos), usando formatos personalizados como RTF(Rich Text Format).

La segunda razón es que los editores de texto son herramientas especiales para editar código, porque tienen características útiles como resaltar el código con diferentes colores de acuerdo a su significado, o cerrar comillas automáticamente.

Veremos todo esto en acción más adelante. Pronto el editor de código se convertirá en una de tus herramientas favoritas. :)

Crea un entorno virtual (virtualenv) e instala Django

> Parte de esta sección se basa en tutoriales por Geek Girls Carrots (https://github.com/ggcarrots/django-carrots).
>
> Parte de este capítulo está basada en el django-marcador tutorial bajo la licencia Creative Commons Attribution-ShareAlike 4.0 internacional. El tutorial de django-marcador tiene derechos de autor de Markus Zapke-Gündemann et al.

Entorno virtual

Antes de instalar Django, instalaremos una herramienta extremadamente útil que ayudará a mantener tu entorno de desarrollo ordenado en tu computadora. Es posible saltarse este paso, pero es altamente recomendable. ¡Empezar con la mejor configuración posible te ahorrará muchos problemas en el futuro!

Así que, vamos a crear un **entorno virtual** (también llamado un *virtualenv*). Virtualenv aísla tu configuración de Python/Django por cada proyecto. Esto quiere decir que cualquier cambio que hagas en un sitio web no afectará a ningún otro que estés desarrollando. Genial, ¿no?

Todo lo que necesitas hacer es encontrar un directorio en el que quieras crear el `virtualenv`; tu directorio home, por ejemplo. En Windows, puede verse como `C:\Users\Name` (donde `Name` es el nombre de tu usuario).

> **NOTA:** En Windows, asegúrate de que este directorio no contiene caracteres especiales o acentuados; si tu nombre de usuario contiene caracteres acentuados, usa un directorio distinto, por ejemplo `C:\djangogirls`.

Para este tutorial usaremos un nuevo directorio `djangogirls` en tu directorio home:

command-line

```
$ mkdir djangogirls
$ cd djangogirls
```

Haremos un virtualenv llamado `myvenv`. El comando general estará en el formato:

command-line

```
$ python3 -m venv myvenv
```

Virtual environment: Windows
Para crear un nuevo `virtualenv`, necesitas abrir una terminal "command prompt" y ejecutar `python -m venv myvenv`. Se verá así:

command-line

```
C:\Users\Name\djangogirls> python -m venv myvenv
```

Donde `myvenv` es el nombre de tu `virtualenv`. Puedes utilizar cualquier otro nombre, pero asegúrate de usar minúsculas y no usar espacios, acentos o caracteres especiales. También es una buena idea mantener el nombre corto. ¡Vas utilizarlo muchas vecesl!

Virtual environment: Linux and OS X
Podemos crear un `virtualenv` en Linux y OS X, es tan sencillo como ejecutar `python3 -m venv myvenv`. Se verá así:

command-line

```
$ python3 -m venv myvenv
```

`myvenv` es el nombre de tu `virtualenv`. Puedes usar cualquier otro nombre, pero sólo utiliza minúsculas y no incluyas espacios. También es una buena idea mantener el nombre corto. ¡Vas a referirte muchas veces a él!

> **NOTA:** En algunas versiones de Debian/Ubuntu, puede que obtengas el siguiente error:
>
> command-line
>
> ```
> The virtual environment was not created successfully because ensurepip is not available.  En sistemas Debian/U
> buntu, tendrás que instalar el paquete python3-venv usando el siguiente comando.
>  apt-get install python3-venv
> Puede que tengas que usar sudo con este comando.  Después de instalar el paquete python3-venv, vuelve a crear
> tu entorno virtual.
> ```
>
> En este caso, sigue las instrucciones anteriores e instala el paquete `python3-venv`:
>
> command-line
>
> ```
> $ sudo apt install python3-venv
> ```
>
> **NOTA:** En algunas versiones de Debian/Ubuntu inicializar el entorno virtual de esta manera da el siguiente error:
>
> command-line
>
> ```
> Error: Command '['/home/eddie/Slask/tmp/venv/bin/python3', '-Im', 'ensurepip', '--upgrade', '--default-pip']'
> returned non-zero exit status 1
> ```
>
> Para evitar esto, utiliza directamente el comando `virtualenv`.
>
> command-line
>
> ```
> $ sudo apt-get install python-virtualenv
> $ virtualenv --python=python3.6 myvenv
> ```
>
> **NOTA:** Si obtienes un error como
>
> command-line
>
> ```
> E: Unable to locate package python3-venv
> ```
>
> entonces ejecuta:
>
> command-line
>
> ```
> sudo apt install python3.6-venv
> ```

Trabajar con virtualenv

El comando anterior creará un directorio llamado `myvenv` (o cualquier nombre que hayas elegido) que contiene nuestro entorno virtual (básicamente un montón de archivos y carpetas).

Working with virtualenv: Windows
Inicia el entorno virtual ejecutando:

command-line

```
C:\Users\Name\djangogirls> myvenv\Scripts\activate
```

> **Nota:** en 10 de Windows puedes obtener un error en Windows PowerShell que dice `execution of scripts is disabled on this system`. En este caso, abre otro Windows PowerShell con la opción "Ejecutar como administrador". Luego intenta escribir el siguiente comando antes de inicializar tu entorno virtual:
>
> command-line
>
> ```
> C:\WINDOWS\system32 > Set-ExecutionPolicy -ExecutionPolicy RemoteSigned
>  Execution Policy Change
>  The execution policy helps protect you from scripts that you do not trust. Changing the execution policy mi
> ght expose you to the security risks described in the about_Execution_Policies help topic at http://go.microso
> ft.com/fwlink/?LinkID=135170. Do you want to change the execution policy? [Y] Yes  [A] Yes to All  [N] No  [L]
>  No to All  [S] Suspend  [?] Help (default is "N"): A
> ```

Working with virtualenv: Linux and OS X
Inicia el entorno virtual ejecutando:

command-line

```
$ source myvenv/bin/activate
```

¡Recuerda reemplazar `myvenv` con tu nombre de `virtualenv` que hayas elegido!

> **NOTA:** a veces `source` podría no estar disponible. En ese caso trata hacerlo de esta forma:
>
> command-line
>
> ```
> $ . myvenv/bin/activate
> ```

Sabrás que tienes `virtualenv` iniciado cuando veas que la línea de comando en tu consola tiene el prefijo `(myvenv)`.

Cuando trabajes en un entorno virtual, `python` automáticamente se referirá a la versión correcta, de modo que puedes utilizar `python` en vez de `python3`.

Ok, tenemos todas las dependencias importantes en su lugar. ¡Finalmente podemos instalar Django!

Instalar Django

Ahora que tienes tu `virtualenv` iniciado, puedes instalar Django.

Antes de hacer eso, debemos asegurarnos que tenemos la última versión de `pip`, el software que utilizamos para instalar Django:

command-line

```
(myvenv) ~$ python3 -m pip install --upgrade pip
```

Instalar paquetes con un fichero de requisitos (requirements)

Un fichero de requisitos (requirements) tiene una lista de dependencias que se deben instalar mediante `pip install`:

Lo primero, crea un fichero `requirements.txt` dentro del directorio `djangogirls/`, usando el editor de código que hemos instalado hace un momento:

```
djangogirls
└──requirements.txt
```

Dentro del fichero `djangogirls/requirements.txt` deberías tener el siguiente texto:

djangogirls/requirements.txt

```
Django~=2.0.6
```

Ahora, ejecuta `pip install -r requirements.txt` para instalar Django.

command-line

```
(myvenv) ~$ pip install -r requirements.txt
Collecting Django~=2.0.6 (from -r requirements.txt (line 1))
  Downloading Django-2.0.6-py3-none-any.whl (7.1MB)
Installing collected packages: Django
Successfully installed Django-2.0.6
```

Installing Django: Windows

> Si obtienes un error al ejecutar pip en Windows comprueba si la ruta de tu proyecto contiene espacios, acentos o caracteres especiales (por ejemplo, `C:\Users\User Name\djangogirls`). Si los tiene, por favor considera moverla a otro lugar sin espacios, acentos o caracteres especiales (sugerencia: `C:\djangogirls`). Crea un nuevo virtualenv en el nuevo directorio, luego borra el viejo y reintenta el comando anterior. (Mover el directorio de virtualenv no funciona puesto que virtualenv usa rutas absolutas.)

Installing Django: Windows 8 and Windows 10

> Puede que tu línea de comandos se congele después de que intentes instalar Django. Si esto sucede, en vez del comando anterior utiliza:
>
> command-line
>
> ```
> C:\Users\Name\djangogirls> python -m pip install -r requirements.txt
> ```

Installing Django: Linux

> Si obtienes un error al ejecutar pip en Ubuntu 12.04 ejecuta `python -m pip install -U --force-reinstall pip` para arreglar la instalación de pip en el virtualenv.

¡Eso es todo! Ahora estás lista (por fin) para crear una aplicación Django!

Instalar Git

Git es un "sistema de control de versiones" que utilizan muchos programadores. Este software puede seguir los cambios realizados en archivos a lo largo del tiempo de forma que más tarde puedas volver a cualquier versión anterior. Es un poco parecido a la opción de "control de cambios" de Microsoft Word, pero mucho más potente.

Instalar Git

Installing Git: Windows

Puedes descargar Git desde git-scm.com. Puedes hacer click en "next" en todos los pasos a excepción de uno; en el quinto paso titulado "Adjusting your PATH environment", escoge "Use Git and optional Unix tools from the Windows Command Prompt" (la de abajo del todo). Aparte de eso, los valores por defecto están bien. "Checkout Windows-style, commit Unix-style line endings" también está bien.

No olvides reiniciar la terminal o powershell después de que la instalación termine.

Installing Git: OS X

Descarga Git de git-scm.com y sigue las instrucciones.

> **Nota** Si estas usando OS X 10.6, 10.7 o 10.8, tendrás que instalar git desde aquí: Git installer for OS X Snow Leopard

Installing Git: Debian or Ubuntu

command-line

```
$ sudo apt install git
```

Installing Git: Fedora

command-line

```
$ sudo dnf install git
```

Installing Git: openSUSE

command-line

```
$sudo zypper install git
```

Crear una cuenta de GitHub

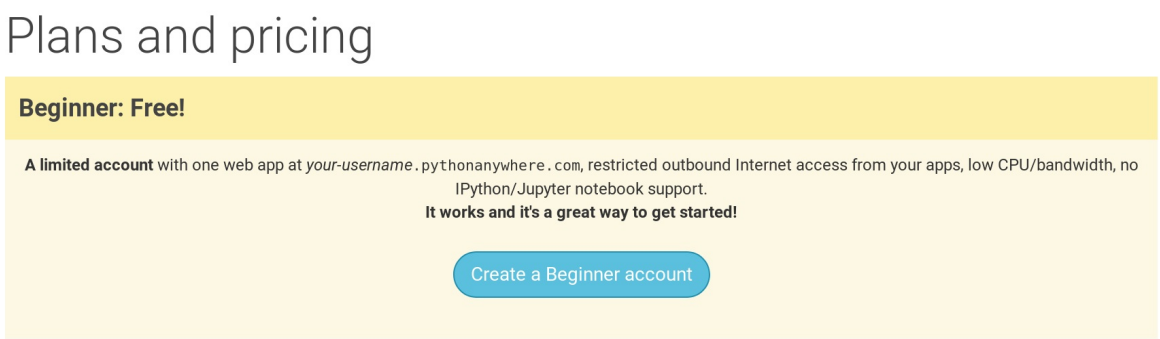
Visita GitHub.com y registra una nueva cuenta de usuario gratuita.

Crear una cuenta de PythonAnywhere

PythonAnywhere es un servicio para ejecutar código Python en servidores "en la nube". Lo vamos a usar para alojar nuestro sitio para que esté disponible en Internet.

Crea una cuenta de Principiante ("Beginner") en PythonAnywhere (el nivel gratuito está bien, no necesitas tarjeta de crédito).

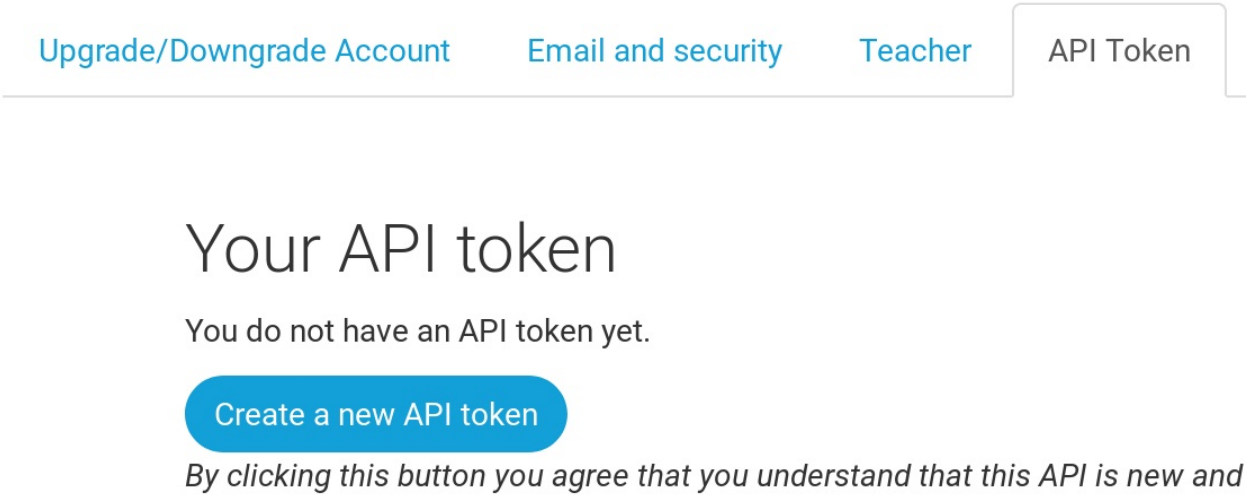
- www.pythonanywhere.com



> **Nota** Cuando elijas un nombre de usuario, recuerda que la URL de tu blog tendrá la forma `tunombredeusuario.pythonanywhere.com`, así que lo mejor será usar tu apodo o elegir un nombre que indique de qué trata tu blog.

Crear un token para la API de PythonAnywhere

Solo tendrás que hacer esto la primera vez. Cuando entres en PythonAnywhere, verás el panel de control ("dashboard"). En la esquina superior derecha, busca el enlace la página de tu cuenta ("Account"), selecciona una pestaña llamada "API token" y pulsa el botón que pone "Create new API token".



Comienza a leer

Felicitaciones, ya tienes todo configurado y listo para seguir! Si aún tienes tiempo antes del taller, sería útil comenzar a leer algunos de los capítulos iniciales:

- ¿Cómo funciona internet?
- Introducción a la línea de comandos
- Introducción a Python
- ¿Qué es Django?

¡Disfrutar el taller!

Al comenzar el taller, podrás ir directamente a Tu primer proyecto en Django! porque ya cubriste el material de los capítulos anteriores.

Configuración de Chromebook

> **Nota** Si ya ejecutaste todas las instrucciones indicadas durante la instalación, no es necesario volverlo a hacer - puedes omitir lo que sigue hasta la Introducción a Python.

Puedes saltarte esta sección en caso de que no estés usando un Chromebook. Si lo estás usando, tu experiencia de instalación será algo diferente. Puedes ignorar el resto de las instrucciones de instalación.

IDE en la nube (PaizaCloud Cloud IDE, AWS Cloud9)

Un IDE en la nube es una herramienta que te da un editor de código y acceso a un ordenador conectado a internet en el que puedes instalar, escribir y ejecutar software. En este tutorial, el IDE en la nube te servirá como tu *máquina local*. Seguirás ejecutando comandos en una terminal igual que tus compañeros de clase en OS X, Ubuntu, o Windows, pero tu terminal en realidad estará conectada a un ordenador corriendo en otro sitio que el IDE en la nube gestionará para ti. Aquí están las instrucciones para IDEs en la nube (PaizaCloud Cloud IDE, AWS Cloud9). Puedes elegir uno de los IDEs en la nube, y seguir sus instrucciones.

PaizaCloud Cloud IDE

1. Ve a PaizaCloud Cloud IDE
2. Crea una cuenta
3. Haz click en *New Server*
4. Haz click en el botón Terminal (en el lado izquierdo de la ventana)

Ahora deberías ver una interfaz con una barra y botones en la izquierda. Haz click en al botón "Terminal" para abrir la ventana de la terminal con un símbolo de sistema como este:

Terminal

```
$
```

La terminal enviará tus instrucciones al ordenador que Cloud 9 ha preparado para ti. Puedes redimensionar o maximizar la ventana para hacerla un poco mas grande.

AWS Cloud9

1. Ve a AWS Cloud9
2. Crea una cuenta
3. Haz click en *Create Environment*

Deberías ver un interfaz con una barra lateral, una ventana principal grande con texto y una ventana pequeña abajo del todo parecida a esto:

bash

```
yourusername:~/workspace $
```

La parte inferior es tu *terminal*, donde escribirás las instrucciones para el ordenador que Cloud 9 te ha preparado. Puedes redimensionar la ventana para hacerla un poco más grande.

Entorno Virtual

Un entorno virtual (también llamado virtualenv) es como una caja privada donde podemos guardar código útil para el proyecto en el que estamos trabajando. Lo usamos para guardar por separado los trozos de código de distintos proyectos, y que así, las cosas no se mezclen entre proyectos.

En la terminal de la parte inferior de Cloud 9, ejecuta lo siguiente:

Cloud 9

```
sudo apt update
sudo apt install python3.6-venv
```

Si aun así, no te funciona, pide ayuda a tu mentor.

A continuación, ejecuta:

Cloud 9

```
mkdir djangogirls
cd djangogirls
python3.6 -mvenv myvenv
source myvenv/bin/activate
pip install django~=2.0.6
```

(fíjate que en la última línea hemos usado una tilde seguida de un signo de igual: ~=).

GitHub

Hazte una cuenta de GitHub.

Python Anywhere

El tutorial de Django Girls tiene una sección que se llama "Despliegue", que es el proceso de coger el código de tu nueva aplicación web y ponerlo en un ordenador públicamente accesible (un servidor) para que todo el mundo pueda ver tu trabajo.

Esta parte "chirría" un poco cuando haces el tutorial en un Chromebook, porque que ya estamos usando un ordenador que ya está en Internet (en lugar de un ordenador local como un portátil). Sin embargo, aún tiene sentido, si pensamos que nuestro espacio de trabajo en Cloud9 es un lugar para nuestro trabajo "en progreso" y PythonAnywhere es el lugar donde enseñar nuestro sitio web más terminado.

Así que créate una cuenta de PythonAnywhere en www.pythonanywhere.com.

¿Cómo funciona Internet?

> Para lectores en casa: este capítulo está cubierto en el video ¿Cómo funciona Internet?.
>
> Este capitulo está inspirado en la charla "How the Internet works" de Jessica McKellar (http://web.mit.edu/jesstess/www/).

Apostamos a que utilizas Internet todos los días. Pero, ¿sabes lo que pasa cuando escribes una dirección como http://djangogirls.org en tu navegador y presionas `enter`?

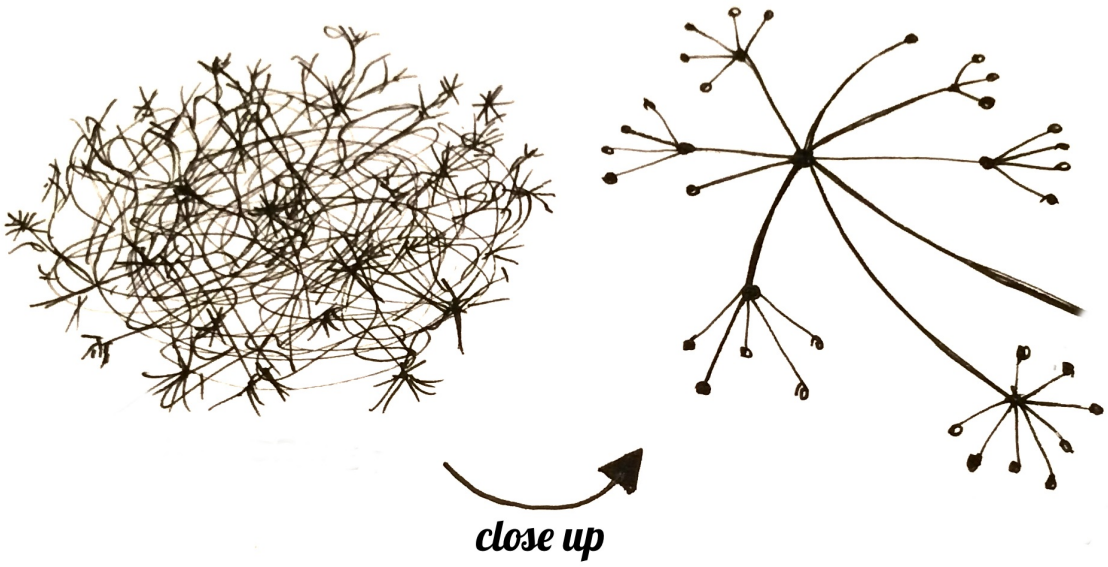
Lo primero que hay que entender es que un sitio web se compone de unos cuantos ficheros almacenados en un disco duro. Al igual que tus películas, música o fotos. Sin embargo, los sitios web poseen una peculiaridad: incluyen un código informático llamado HTML.

Si no estás familiarizada con la programación, puede ser difícil captar HTML a la primera, pero tus navegadores web (como Chrome, Safari, Firefox, etc.) lo aman. Los navegadores están diseñados para entender este código, seguir sus instrucciones y presentar estos archivos de los cuales está hecho tu sitio web, exactamente de la forma que quieres.

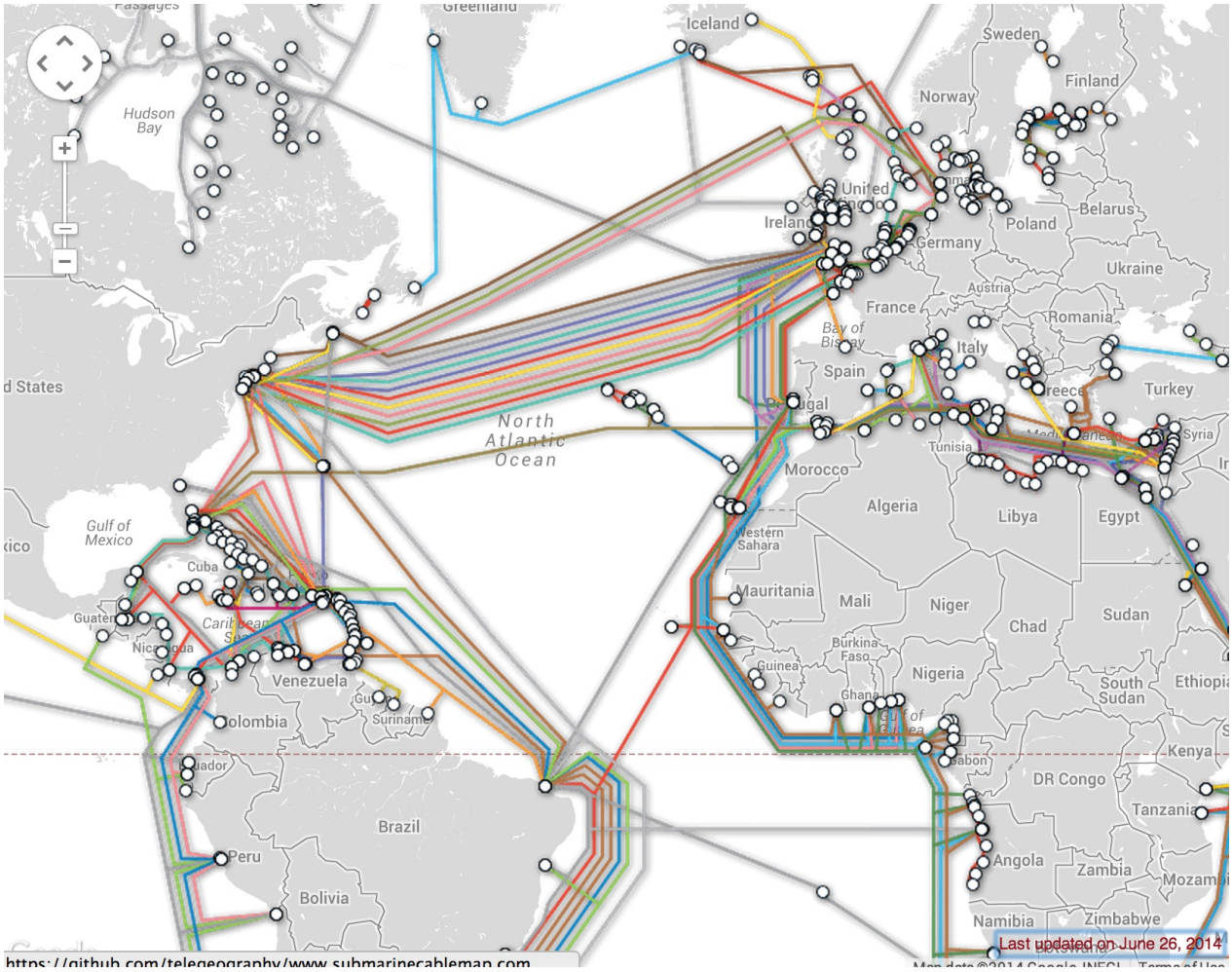
Como cualquier otro archivo, tenemos que guardar los archivos HTML en algún lugar de un disco duro. Para Internet, utilizamos equipos especiales, de gran alcance llamados *servidores*. Estos no tienen una pantalla, ratón o teclado, debido a que su propósito es almacenar datos y servirlos. Por esa razón son llamados *servidores* -- porque *sirven* los datos.

OK, pero quieres saber cómo Internet se ve, ¿cierto?

¡Te hemos hecho una imagen! Luce algo así:

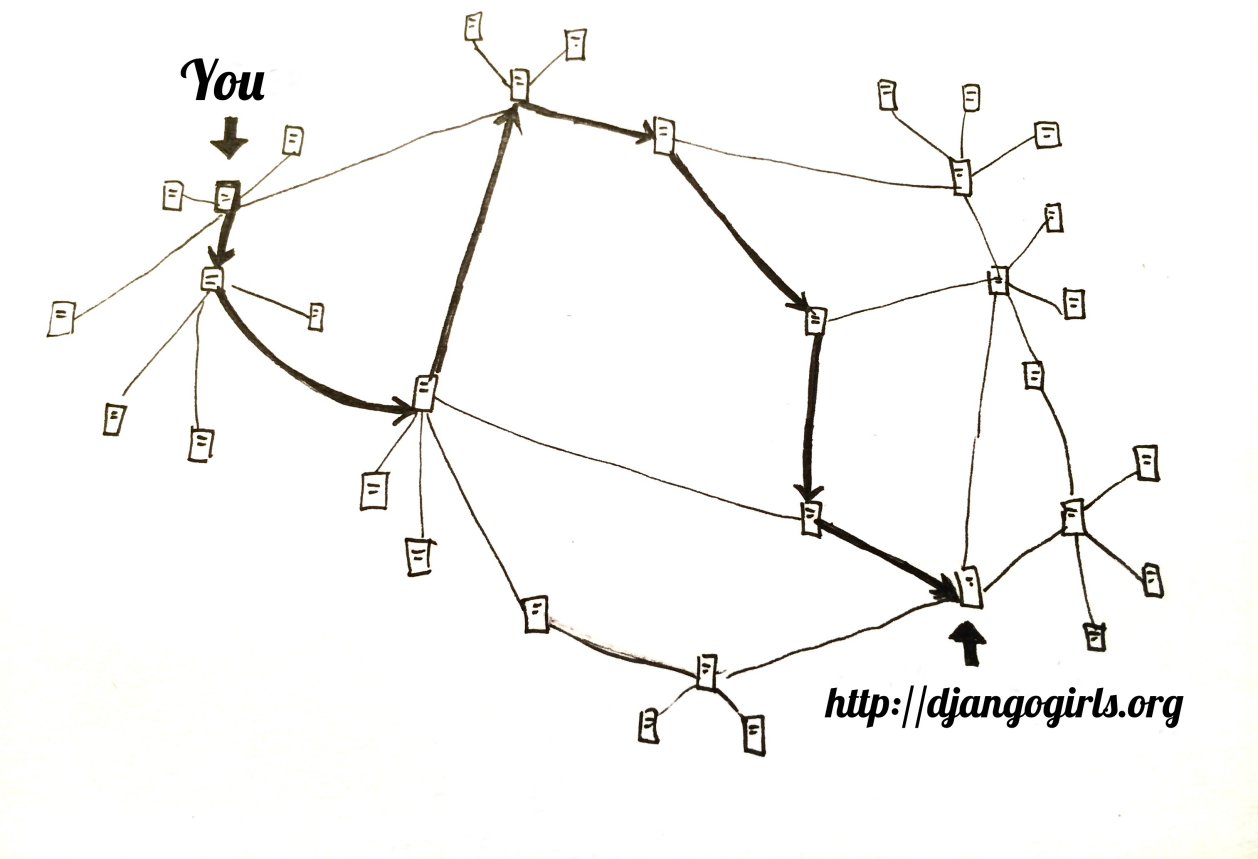


Parece un lío, ¿no? De hecho, es una red de máquinas interconectadas (los *servidores* que nombramos anteriormente). ¡Cientos de miles de máquinas! ¡Muchos, muchos kilómetros de cables alrededor del mundo! Puedes visitar el sitio web Submarine Cable Map (http://submarinecablemap.com/) y ver lo complicada que es la red. Aquí hay una captura de pantalla de la página web:



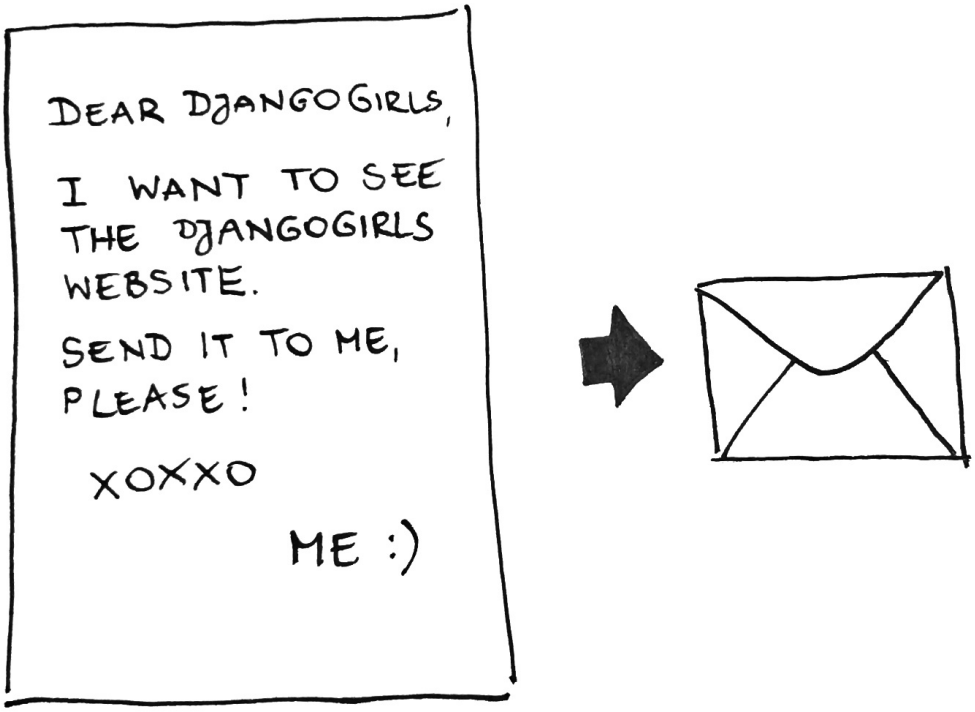
Es fascinante, ¿no? Pero sería imposible tener un cable entre todas y cada una de las máquinas conectadas a internet. Así que, para llegar a una máquina (por ejemplo la que aloja http://djangogirls.org) tenemos que elevar una solicitud mediante una gran cantidad de máquinas diferentes.

Se parece a esto:



Imagina que cuando escribes http://djangogirls.org, estas enviando una carta que dice: "Estimadas Django Girls, me gustaría ver su sitio web djangogirls.org. Por favor, envíenmelo!"

Tu carta va hacia la oficina de correo más cercana. Luego va a otra que es un poco más cerca de su destinatario, luego otra y otra hasta que es entregada a su destino. La única cosa diferente es que si envías muchas cartas (*paquetes de datos*) al mismo lugar, cada una podría ir a través de oficinas de correos totalmente distintas (*routers*). Esto depende de cómo se distribuyen en cada oficina.



Así es como funciona - se envían mensajes y se espera una respuesta. En lugar de papel y lápiz, se usan bytes de datos, pero ¡la idea es la misma!

En lugar de direcciones con el nombre de la calle, ciudad, código postal y nombre del país, utilizamos direcciones IP. Tu computadora pide primero el DNS (Domain Name System - en español Sistema de Nombres de Dominio) para traducir djangogirls.org a una dirección IP. Funciona en cierta manera como los viejos directorios telefónicos donde puedes buscar el nombre de la persona que se deseas contactar y encontrar su número de teléfono y dirección.

Cuando envías una carta, ésta necesita tener ciertas características para ser entregada correctamente: una dirección, sello, etc. También utilizas un lenguaje que el receptor pueda entender, ¿cierto? Lo mismo se aplica a los *paquetes de datos* que envía para ver un sitio Web. Utilizamos un protocolo llamado HTTP (Protocolo de transferencia de hipertexto).

Así que, básicamente, cuando tienes un sitio web necesitas tener un *servidor* (la máquina) donde este vive. Cuando el *servidor* recibe una *petición* entrante (en una carta), este envía su sitio de Internet (en otra carta).

Ya que este es un tutorial de Django, puede que te preguntes que lo que Django hace. Bueno, cuando envías una respuesta, no siempre quieres enviar lo mismo a todo el mundo. Es mucho mejor si tus cartas son personalizadas, especialmente para la persona que acaba de escribir, ¿cierto? Django nos ayuda con la creación de estas cartas personalizadas. :)

Suficiente conversación - tiempo de crear!

Introducción a la interfaz de línea de comandos

> Para los lectores en casa: este capitulo es visto en el video Tu nuevo amigo: línea de comandos.

Es emocionante, ¿verdad? Vas a escribir tu primera línea de código en pocos minutos! :)

Permítenos presentarte a tu primer nuevo amigo: ¡la línea de comandos!

Los siguientes pasos te mostrarán cómo usar aquella ventana negra que todos los hackers usan. Puede parecer un poco aterrador al principio pero es solo un mensaje en pantalla que espera a que le des órdenes.

> **Nota** Ten en cuenta que a lo largo de este libro usamos los términos 'directorio' y 'carpeta' indistintamente pero son la misma cosa.

¿Qué es la línea de comandos?

La ventana, que generalmente es llamada **línea de comandos** ó **interfaz de línea de comandos**, es una aplicación basada en texto para ver, manejar y manipular archivos en tu ordenador. Similar a Windows Explorer o Finder en Mac, pero sin la interfaz gráfica. Otros nombres para la línea de comandos son: *cmd*, *CLI*, *prompt* -símbolo de sistema-, *console* -consola- o *terminal*.

Abrir la interfaz de línea de comandos

Lo primero que debemos hacer para empezar a experimentar con nuestra interfaz de línea de comandos es abrirla.

Opening: Windows
Ve a Menú de inicio → Windows System → Command Prompt.

> En versiones anteriores de Windows, busca en Menú de inicio → Todos los programas → Accesorios → Command Prompt.

Opening: OS X
Ve a Aplicaciones → Utilidades → Terminal.

Opening: Linux
Probablemente estará en Aplicaciones → Accesorios → Terminal, pero en tu sistema puede estar en un sitio distinto. Si no lo encuentras, busca en Google. :)

Símbolo del Sistema (Prompt)

Ahora deberías ver una ventana blanca o negra que está esperando tus órdenes.

Prompt: OS X and Linux
Si estás en un Mac o Linux, seguramente verás un símbolo `$`, como este:

command-line

```
$
```

Prompt: Windows
En Windows, es un signo así `>`, como este:

command-line

```
>
```

Cada comando será precedido por este signo y un espacio, pero no tienes que escribirlo. Tu computadora lo hará por ti. :)

> Sólo una pequeña nota: en tu caso puede que haya algo como `C:\Users\ola>` o `Olas-MacBook-Air:~ ola$` antes del símbolo prompt y eso está perfecto.

La parte hasta e incluyendo `$` o `>` se llama la *línea de comandos* o *prompt*. Esta te solicita escribir algo ahí.

En el tutorial, cuando queremos escribir un comando, incluiremos el `$` o `>` y de vez en cuando más a la izquierda. Ignora la parte izquierda y teclea sólo el comando, que es lo que empieza después del prompt.

Tu primer comando (¡BIEN!)

Comencemos escribiendo este comando:

Your first command: OS X and Linux

command-line

```
$ whoami
```

Your first command: Windows

command-line

```
> whoami
```

Y pulsa `enter`. Este es nuestro resultado:

command-line

```
$ whoami olasitarska
```

Como puedes ver, el ordenador ha imprimido tu nombre de usuario. Genial, ¿eh? :)

> Trata de escribir cada comando, no copies y pegues. ¡Te acordarás más de esta manera!

Fundamentos

Cada sistema operativo tiene un conjunto diferente de comandos para la línea de comandos, así que asegúrate de seguir las instrucciones para tu sistema operativo. Vamos a intentarlo, ¿de acuerdo?

Directorio actual

Estaría bien saber dónde estamos ahora, ¿verdad? Vamos a ver. Escribe este comando y pulsa `intro`:

Current directory: OS X and Linux

command-line

```
$ pwd
/Users/olasitarska
```

> Nota: 'pwd' significa 'print working directory' - en español, 'mostrar directorio de trabajo'.

Current directory: Windows

command-line

```
> cd
C:\Users\olasitarska
```

> Nota: 'cd' significa 'cambiar directorio'. Con powershell se puede utilizar pwd al igual que en Linux o Mac OS X.

Probablemente verás algo similar en tu máquina. Una vez que abres la línea de comandos generalmente empiezas en el directorio home de tu usuario.

Listar ficheros y directorios

¿Qué hay aquí? Sería bueno saber. Veamos:

List files and directories: OS X and Linux

command-line

```
$ ls
Applications
Desktop
Downloads
Music
...
```

List files and directories: Windows

command-line

```
> dir
Directory of C:\Users\olasitarska
05/08/2014 07:28 PM <DIR> Applications
05/08/2014 07:28 PM <DIR> Desktop
05/08/2014 07:28 PM <DIR> Downloads
05/08/2014 07:28 PM <DIR> Music
...
```

> Nota: En powershell también puedes utilizar 'ls' como en Linux y Mac OS X.

Cambia el directorio actual

Ahora, vayamos a nuestro directorio Desktop, el escritorio:

Change current directory: OS X and Linux

command-line

```
$ cd Desktop
```

Change current directory: Windows

command-line

```
> cd Desktop
```

Comprueba si realmente ha cambiado:

Check if changed: OS X and Linux

command-line

```
$ pwd
/Users/olasitarska/Desktop
```

Check if changed: Windows

command-line

```
> cd
C:\Users\olasitarska\Desktop
```

¡Aquí está!

> Truco pro: si escribes `cd D` y luego pulsas `tab` en el teclado, la línea de comandos automáticamente completará el resto del nombre para que puedas navegar más rápido. Si hay más de una carpeta que empiece con "D", dale al botón `tab` dos veces para obtener una lista de opciones.

Crear directorio

¿Qué tal si creamos un directorio de práctica en el escritorio? Lo puedes hacer de esta manera:

Create directory: OS X and Linux

command-line

```
$ mkdir practice
```

Create directory: Windows

command-line

```
> mkdir practice
```

Este pequeño comando creará una carpeta con el nombre `practice` en el escritorio. Puedes comprobar que efectivamente está allí mirando en tu Escritorio o ejecutando un comando `ls` o `dir`. ¡Inténtalo! :)

> Truco pro: Si no quieres escribir una y otra vez los mismos comandos, prueba pulsando la `flecha arriba` y la `flecha abajo` de tu teclado para ir pasando por los comandos utilizados recientemente.

¡Ejercicios!

Un pequeño reto para ti: en el directorio `practice` que acabas de crear crea un directorio llamado `test`. (Utiliza los comandos `cd` y `mkdir`.)

Solución:

Exercise solution: OS X and Linux

command-line

```
$ cd practice
$ mkdir test
$ ls
test
```

Exercise solution: Windows

command-line

```
> cd practice
> mkdir test
> dir
05/08/2014 07:28 PM <DIR> test
```

¡Enhorabuena! :)

Limpieza

No queremos dejar un lío, así que vamos a eliminar todo lo que hemos hecho hasta este momento.

En primer lugar, tenemos que volver al escritorio:

Clean up: OS X and Linux

command-line

```
$ cd ..
```

Clean up: Windows

command-line

```
> cd ..
```

Usar `..` con el comando `cd` hará que cambie el directorio actual al directorio padre (el que contiene el directorio actual).

Revisa dónde estás:

Check location: OS X and Linux

command-line

```
$ pwd
/Users/olasitarska/Desktop
```

Check location: Windows

command-line

```
> cd
C:\Users\olasitarska\Desktop
```

Es el momento de eliminar el directorio `practice`:

> **Atención**: Eliminar archivos utilizando `del`, `rmdir` o `rm` hace que no puedan recuperarse, lo que significa que los *archivos borrados desaparecerán para siempre*! Así que ten mucho cuidado con este comando.

Delete directory: Windows Powershell, OS X and Linux

command-line

```
$ rm -r practice
```

Delete directory: Windows Command Prompt

command-line

```
> rmdir /S practice
practice, Are you sure <Y/N>? Y
```

¡Hecho! Para asegurarnos de que realmente se ha eliminado, vamos a comprobarlo:

Check deletion: OS X and Linux

command-line

```
$ ls
```

Check deletion: Windows

command-line

```
> dir
```

Salida

Esto es todo por ahora! Ya puedes cerrar la línea de comandos sin problema. Vamos a hacerlo al estilo hacker, ¿vale? :)

Exit: OS X and Linux

command-line

```
$ exit
```

Exit: Windows

command-line

```
> exit
```

Genial, ¿no? :)

Resumen

Aquí hay una lista de algunos comandos útiles:

Comando (Windows)	Comando (Mac OS / Linux)	Descripción	Ejemplo
salida	salida	Cierra la ventana	**salida**
cd	cd	Cambia el directorio	**cd test**
cd	pwd	Mostrar el directorio actual	**cd** (Windows) o **pwd** (Mac OS / Linux)
dir	ls	Lista directorios/archivos	**dir**
copy	cp	Copia de archivos	**copy c:\test\test.txt c:\windows\test.txt**
move	mv	Mueve archivos	**move c:\test\test.txt c:\windows\test.txt**
mkdir	mkdir	Crea un nuevo directorio	**mkdir testdirectory**
rmdir (o del)	rm	Eliminar un archivo	**del c:\test\test.txt**
rmdir /S	rm -r	Eliminar un Directorio	**rm -r testdirectory**

Estos son sólo unos pocos de los comandos que se pueden ejecutar en la línea de comandos, pero hoy no vas a utilizar ninguno más.

Si tienes curiosidad, ss64.com contiene una referencia completa de comandos para todos los sistemas operativos.

¿Listo?

¡Vamos a sumergirnos en Python!

Vamos a empezar con Python

¡Por fin estamos aquí!

Pero primero, déjanos decirte qué es Python. Python es un lenguaje de programación muy popular que puede ser usado para crear sitios web, juegos, software científico, gráficos, y más, mucho más.

Python se originó en la década de 1980 y su principal objetivo es ser legible por los seres humanos (¡no sólo por máquinas!). Por esta razón parece mucho más sencillo que otros lenguajes de programación, pero no te preocupes – ¡Python también es realmente poderoso!

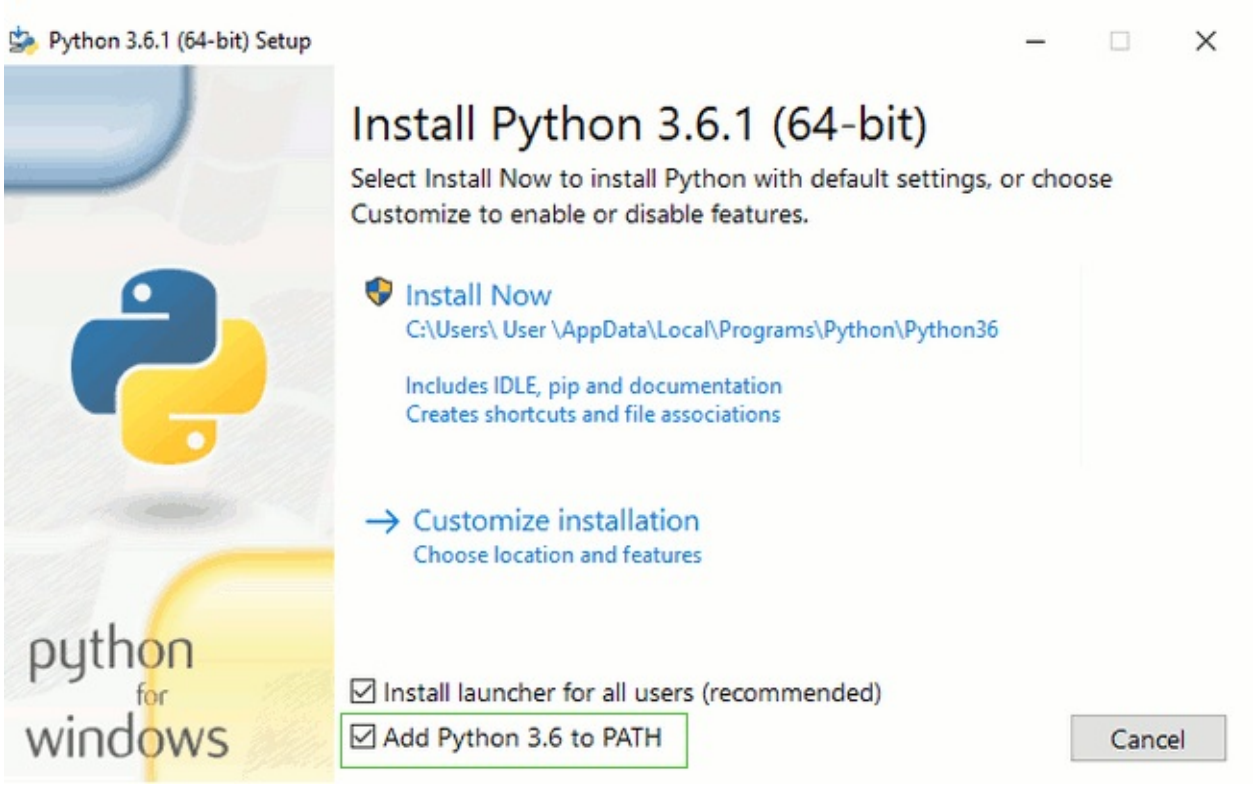
Instalación de Python

> **Nota** Si usas un Chromebook, omite este capítulo y asegúrate de seguir las instrucciones de Chromebook Setup.
>
> **Nota** Si ya has seguido los pasos de instalación, no hay necesidad de hacerlo nuevamente - ¡puedes saltar y continuar al siguiente capítulo!
>
> Para lectores en casa: este capitulo se cubre en el video Installing Python & Code Editor.
>
> Esta sección está basada en un tutorial de Geek Girls Carrots (https://github.com/ggcarrots/django-carrots)

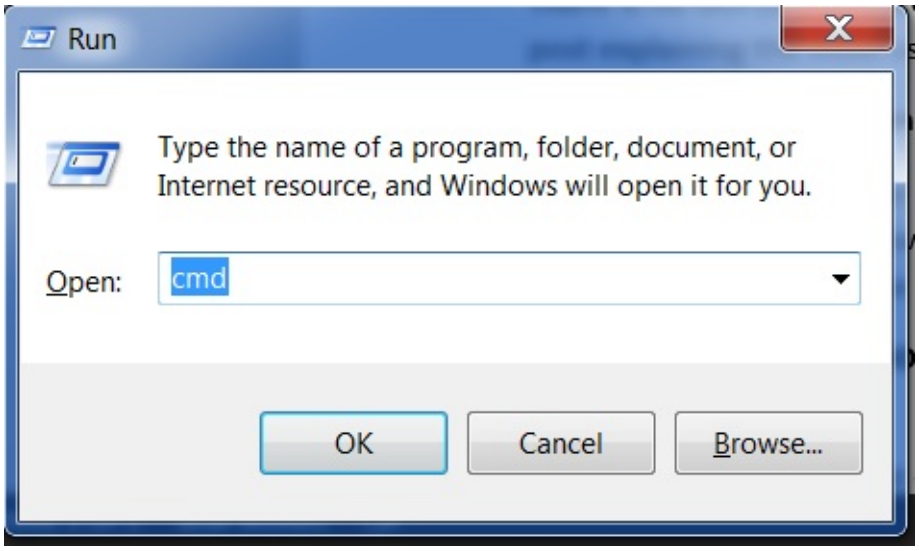
Django esta escrito en Python. Necesitamos Python para hacer cualquier cosa en Django. Comencemos instalandolo!. Tenemos que instalar Python 3.6, así que si tienes una versión anterior, debes actualizarla.

Install Python: Windows
Primero comprueba si tu computador tiene una version de Windows de 32-bit o 64-bit, presiona tecla Windows + Pause/Break, esto abrirá tu System info (información de tu sistema), ahora busta la linea "System type". Puedes descargar Python para Windows desde el sitio web https://www.python.org/downloads/windows/. Haz click en el link "Latest Python 3 release - Python x.x.x.". Si tu computador tiene una versión de Windows de **64 bits**, descarga **Windows x86-64 executable installer**. De lo contrario, descarga **Windows x86 executable installer**. Después de descargar el instalador, debes ejecutarlo (hazle doble click) y sigue las instrucciones.

Algo a tener en cuenta: Durante la instalación notarás una ventana llamada "Setup". Asegúrate de seleccionar la casilla "Add Python 3.6 to Path" y luego haz click en "Install Now", como se muestra a continuación:



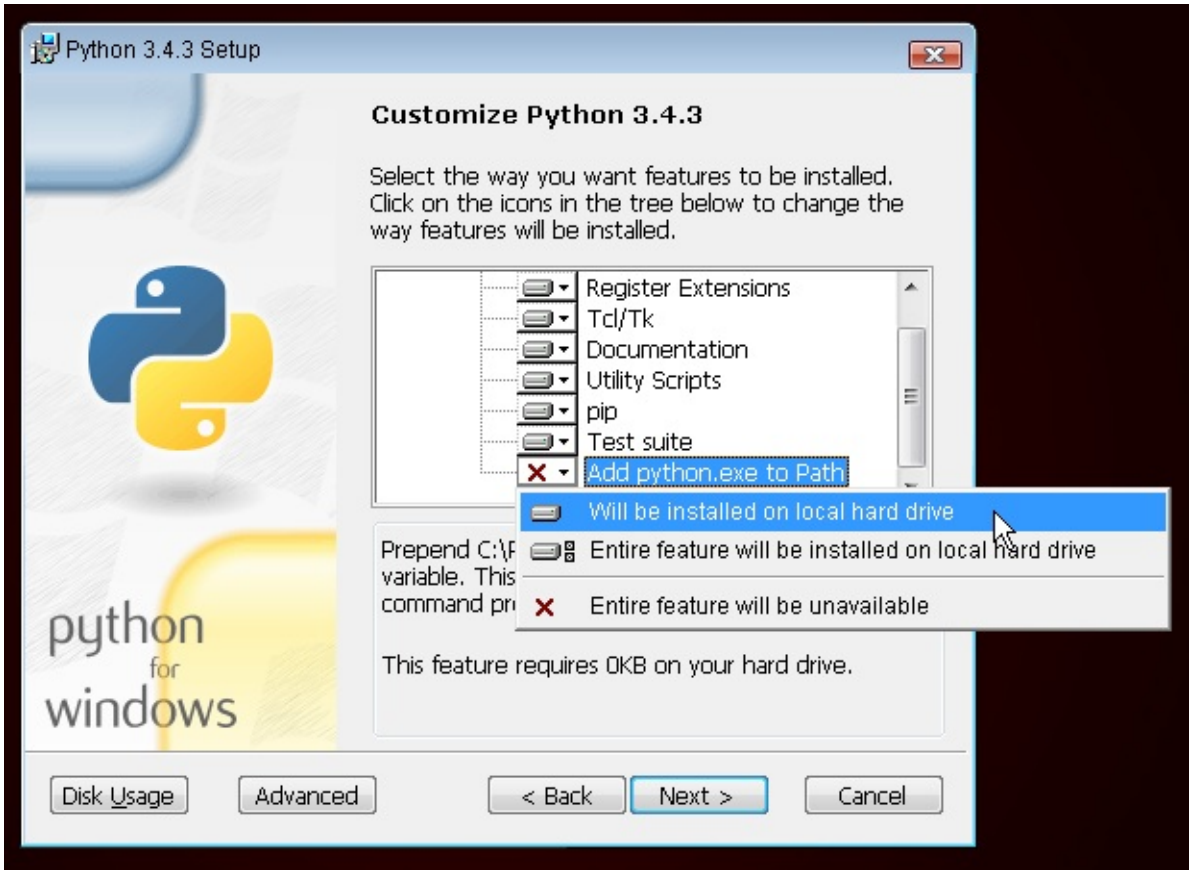
En los próximos pasos, vas a utilizar la línea de comandos de Windows (acerca de la cual también de contaremos algo). De momento, si tienes que teclear algunos comandos, ve al menú de Inicio y teclea "Command Prompt" en el cuadro de búsqueda. (En versiones anteriores de Windows, puedes arrancar la linea de comandos con menú de Inicio → Sistema Windows → Linea de Comandos.) También puedes pulsar la tecla "Windows" + R hasta que aparezca la ventana "Ejecutar" (Run). Para abrir la línea de comandos, escribe "cmd" y pulsa enter en la ventana "Run".



Nota: Si estas utilizando una versión anterior de Windows (7, Vista, o cualquier versión anterior) y el instalador de Python 3.6.x falla con un error, puedes tratar ya sea:

1. instalar todas las actualizaciones de Windows e intentar instalar Python 3.6 nuevamente; o
2. instalar una versión de Python anterior, por ejemplo, 3.4.6.

Si instalas una versión anterior de Python, la pantalla de instalación puede ser un poco diferente a la que se muestra arriba. Asegúrate de desplazarte hacia abajo para ver "Add python.exe to Path", y dar click en el botón a la izquierda y seleccionar "Will be installed on local hard drive":



Install Python: OS X

> **Nota** Antes de instalar Python en OS X, debes asegurarte de que la configuración del Mac permita instalar paquetes que no estén en la App Store. ve a preferencias del sistema (System Preferences, está en la carpeta Aplicaciones), da click en "Seguridad y privacidad" (Security & Privacy) y luego la pestaña "General". Si tu "Permitir aplicaciones descargadas desde:" (Allow apps downloaded from:) está establecida a "Mac App Store," cambia a "Mac App Store y desarrolladores identificados." (Mac App Store and identified developers)

Debes ir al sitio web https://www.python.org/downloads/release/python-361/ y descargar el instalador de Python:

- Descargar el archivo *Mac OS X 64-bit/32-bit installer*,
- Haz doble clic en *python-3.4.3-macosx10.6.pkg* para ejecutar al instalador.

Install Python: Linux

Es muy posible que ya tengas Python instalado de serie. Para verificar que ya lo tienes instalado (y qué versión es), abre una consola y escribe el siguiente comando:

command-line

```
$ python3 --version
Python 3.6.1
```

Si tienes una 'micro versión' diferente de Python instalada, por ejemplo 3.6.0, entonces no tienes que actualizar. Si no tienes instalado Python o si deseas una versión diferente, puedes instalarla de la siguiente manera:

Install Python: Debian or Ubuntu

Escribe este comando en tu consola:

command-line

```
$ sudo apt install python3.6
```

Install Python: Fedora

Usa este comando en tu consola:

command-line

```
$ sudo dnf instalar python3
```

En versiones anteriores de Fedora tal vez te salga un error de que no se encuentra el comando `dnf`. En ese caso utiliza `yum` en su lugar.

Install Python: openSUSE
Usa este comando en tu consola:

command-line

```
$ sudo zypper install python3
```

Verifica que la instalación fue exitosa abriendo una terminal y ejecutando el comando `python3`:

command-line

```
$ python3 --version
Python 3.6.1
```

NOTA: Si estás en Windows y recibes un mensaje de error no se encontró `python3`, intenta usar `python` (sin el `3`) y comprueba si todavía es una versión de Python 3.6.

Si tienes alguna duda o si algo salió mal y no sabes cómo resolverlo - ¡pide ayuda a tu tutor! A veces las cosas no van bien y que es mejor pedir ayuda a alguien con más experiencia.

Editor de código

> Para lectores en casa: este capitulo se explica en el video Installing Python & Code Editor.

Estás a punto de escribir tu primera línea de código, así que ¡ya toca descargar un editor de código!

> **Nota** Si usas un Chromebook, sáltate este capítulo y asegúrate de seguir las instrucciones de Chromebook Setup.
>
> **Nota** Es posible que ya hayas hecho esto en el capítulo de instalación. Si es así, ¡puedes avanzar directamente al siguiente capítulo!

Hay muchos editores diferentes y la elección es principalmente una cuestión de preferencia personal. La mayoría de programadores de Python usan IDEs (Entornos de Desarrollo Integrados) complejos pero muy potentes, como PyCharm. Sin embargo, como principiante, probablemente no es lo más aconsejable; nuestras recomendaciones son igualmente potentes pero mucho más simples.

Abajo presentamos nuestras sugerencias pero, también puedes preguntarle a tu mentor cuáles son las suyas - será más fácil que te ayude.

Gedit

Gedit es un editor de texto de código abierto, gratuito, y que está disponible para todos los sistemas operativos.

Descárgalo aquí

Sublime Text 3

Sublime Text es un editor muy popular con un período de prueba gratis. Es fácil de instalar y de usar, y está disponible para todos los sistemas operativos.

Descárgalo aquí

Atom

Atom es otro editor que usa mucha gente. Es gratuito, de código abierto y hay versiones para Windows, OS X y Linux. Atom ha sido desarrollado por GitHub.

Descárgalo aquí

¿Por qué estamos instalando un editor de texto?

Tal vez te estés preguntando por qué estamos instalando un editor especializado en código, en lugar de usar un editor convencional como Word o Notepad.

La principal razón es que el código tiene que ser **texto plano** y, el problema con programas como Word y Textedit es que no guardan texto plano, sino que guardan texto enriquecido (con estilos), usando formatos personalizados como RTF(Rich Text Format).

La segunda razón es que los editores de texto son herramientas especiales para editar código, porque tienen características útiles como resaltar el código con diferentes colores de acuerdo a su significado, o cerrar comillas automáticamente.

Veremos todo esto en acción más adelante. Pronto el editor de código se convertirá en una de tus herramientas favoritas. :)

Introducción a Python

> Parte de este capítulo se basa en tutoriales de Geek Girls Carrots (https://github.com/ggcarrots/django-carrots).

¡Escribamos algo de código!

La Consola de Python

> Para los lectores en casa: el video conceptos básicos de Python: enteros, cadenas, listas, variables y errores cubre esta parte.

Para empezar a jugar con Python, tenemos que abrir una *línea de comandos* en nuestra computadora. Deberías saber cómo hacerlo, pues lo aprendiste en el capítulo de Introducción a la Línea de Comandos.

Una vez que estés lista, sigue las instrucciones a continuación.

Queremos abrir una consola de Python, así que escribe `python` en Windows o `python3` en Mac OS/Linux y pulsa `intro`.

command-line

```
$ python3
Python 3.6.1 (...)
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

¡Tu primer comando en Python!

Después de ejecutar el comando de Python, el cursor cambiará a `>>>`. Para nosotros esto significa que por ahora sólo podemos utilizar comandos del lenguaje Python. No tienes que escribir e `>>>` pues Python lo hará por ti.

Si deseas salir de la consola de Python en cualquier momento, solo escribe `exit()` o usa el atajo `Ctrl + Z` para Windows y `Ctrl + D` para Mac/Linux. Luego no verás más `>>>`.

Por ahora, no queremos salir de la consola de Python. Deseamos aprender más sobre ella. Vamos a comenzar escribiendo algo de matemática, escribe `2 + 3` y oprime la tecla `enter`.

command-line

```
>>> 2 + 3
5
```

¡Qué bien! ¿Ves cómo salió la respuesta? ¡Python sabe matemática! Puedes probar otros comandos como:

- `4 * 5`
- `5 - 1`
- `40 / 2`

Para realizar una operación exponencial, digamos 2 elevado al cubo, escribimos:

command-line

```
>>> 2 ** 3
8
```

Diviértete con esto por un momento y luego vuelve aquí. :)

Como puedes ver, Python es una gran calculadora. Si te estás preguntando qué más puedes hacer…

Cadena de caracteres

¿Qué tal tu nombre? Escribe tu nombre entre comillas, así:

command-line

```
>>> "Ola"
'Ola'
```

¡Has creado tu primera cadena de texto! La misma es una secuencia de caracteres que puede ser procesada por una computadora. La cadena de texto (o string, en inglés) debe comenzar y terminar con el mismo carácter. Pueden ser comillas simples (`'`) o dobles (`"`) (¡no hay ninguna diferencia!) Las comillas le dicen a Python que lo que está dentro de ellas es una cadena de texto.

Las cadenas pueden estar concatenadas. Prueba esto:

command-line

```
>>> "Hola " + "Ola"
'Hola Ola'
```

También puedes multiplicar las cadenas por un número:

command-line

```
>>> "Ola" * 3
'OlaOlaOla'
```

Si necesitas poner un apóstrofe dentro de una cadena, hay dos formas de hacerlo.

Usar comillas dobles:

command-line

```
>>> "Runnin' down the hill"
"Runnin' down the hill"
```

o escapar el apóstrofe con la diagonal inversa (``):

command-line

```
>>> 'Runnin\' down the hill'
"Runnin' down the hill"
```

Bien, ¿eh? Para ver tu nombre en letras mayúsculas, escribe:

command-line

```
>>> "Ola".upper()
'OLA'
```

¡Acabas de usar el **método** `upper` sobre tu cadena de texto! Un método (como `upper()`) es un conjunto de instrucciones que Python tiene que realizar sobre un objeto determinado (`"Ola"`) una vez que se le invoca.

Si quieres saber el número de letras que contiene tu nombre, ¡también hay una **función** para eso!

command-line

```
>>> len("Ola")
3
```

Te preguntarás ¿por qué a veces se invoca a las funciones con un `.` al final de una cadena (como `"Ola".upper()`) y a veces se invoca a la función colocando la cadena entre paréntesis? Bueno, en algunos casos las funciones pertenecen a los objetos, como `upper()` , que sólo puede ser utilizada sobre cadenas. En este caso, a la función le llamamos **método**. Otra veces, las funciones no pertenecen a ningún objeto específico y pueden ser usadas en diferentes objetos, como `len()` . Esta es la razón de por qué estamos pasando `"Ola"` como un parámetro a la función `len` .

Resumen

Ok, es suficiente sobre las cadenas. Hasta ahora has aprendido sobre:

- **la terminal** - teclear comandos (código) en la terminal de Python resulta en respuestas de Python
- **números y strings** - en Python los números son usados para matemáticas y strings (cadenas de caracteres) para objetos de texto
- **operadores** - como `+` y `*` , combinan valores para producir uno nuevo
- **funciones** - como `upper()` y `len()` , ejecutan acciones sobre los objetos.

Estos son los conocimientos básicos que puedes aprender de cualquier lenguaje de programación. ¿Lista para algo más difícil? ¡Seguro que lo estás!

Errores

Vamos a intentar algo nuevo. ¿Podemos obtener la longitud de un número de la misma manera que pudimos averiguar la longitud de nuestro nombre? Escribe `len(304023)` y pulsa `enter` :

PythonAnywhere command-line

```
>>> len(304023)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: object of type 'int' has no len()
```

¡Pues tenemos nuestro primer error! El icono de es nuestra manera de darte un aviso de que el código que estás ejecutando no funciona como se espera. ¡Cometer errores (incluso algunos intencionales) son una parte importante del aprendizaje!

Dicho error dice que los objetos de tipo "int" (números enteros) no tienen longitud. ¿Qué podemos hacer ahora? ¿Quizás podamos escribir el número como una cadena? Las cadenas tienen longitud, ¿verdad?

command-line

```
>>> len(str(304023))
6
```

¡Funcionó! Hemos utilizado la función `str` dentro de la función `len` . `str()` convierte todo en cadenas de texto.

- La función `str` convierte cosas en cadenas, **strings**
- La función `int` convierte cosas en enteros, **integers**

> Importante: podemos convertir números en texto, pero no necesariamente podemos convertir texto en números - ¿qué sería `int('hello')` ?

Variables

Un concepto importante en la programación son las variables. Una variable no es más que un nombre para algo, de forma que puedas usarlo más tarde. Los programadores usan estas variables para almacenar datos, hacer su código más legible y para no tener que recordar qué es cada cosa.

Supongamos que queremos crear una nueva variable llamada `name` :

command-line

```
>>> name = "Ola"
```

Tecleamos que nombre es igual a Ola.

Como habrás notado, tu programa no devolvió nada como lo hacía antes. Así que ¿cómo sabemos que la variable existe realmente? Escribe `name` y pulsa `intro` :

command-line

```
>>> name
'Ola'
```

¡Genial! ¡Tu primera variable :)! Siempre puedes cambiar a lo que se refiere:

command-line

```
>>> name = "Sonja"
>>> name
'Sonja'
```

También puedes usarla dentro de funciones:

command-line

```
>>> len(name)
5
```

Increíble, ¿verdad? Por supuesto, las variables pueden ser cualquier cosa, ¡también números! Prueba esto:

command-line

```
>>> a = 4
>>> b = 6
>>> a * b
24
```

Pero ¿qué pasa si usamos el nombre equivocado? ¿Puedes adivinar qué pasaría? ¡Vamos a probar!

command-line

```
>>> city = "Tokyo"
>>> ctiy
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'ctiy' is not defined
```

¡Un error! Como puedes ver, Python tiene diferentes tipos de errores y este se llama **NameError**. Python te dará este error si intentas utilizar una variable que no ha sido definida aún. Si más adelante te encuentras con este error, verifica tu código para ver si no has escrito mal una variable.

¡Juega con esto un rato y descubre qué puedes hacer!

La función print

Intenta esto:

command-line

```
>>> name = 'Maria'
>>> name
'Maria'
>>> print(name)
Maria
```

Cuando sólo escribes `name`, el intérprete de Python responde con la *representación* en forma de cadena de la variable 'name', que son las letras M-a-r-i-a, rodeadas de comillas simples ''. Cuando dices `print(name)`, Python va a "imprimir" el contenido de la variable a la pantalla, sin las comillas, que es más claro.

Como veremos después, `print()` también es útil cuando queremos imprimir cosas desde adentro de las funciones, o cuando queremos imprimir cosas en múltiples líneas.

Listas

Además de cadenas y enteros, Python tiene toda clase de tipos de objetos diferentes. Ahora vamos a introducir uno llamado **list**. Las listas son exactamente lo que piensas que son: objetos que son listas de otros objetos. :)

Anímate y crea una lista:

command-line

```
>>> []
[]
```

Sí, esta lista está vacía. No es muy útil, ¿verdad? Vamos a crear una lista de números de lotería. No queremos repetirnos todo el rato, así que la pondremos también en una variable:

command-line

```
>>> lottery = [3, 42, 12, 19, 30, 59]
```

Muy bien, ¡tenemos una lista! ¿Qué podemos hacer con ella? Vamos a ver cuántos números de lotería hay en la lista. ¿Tienes alguna idea de qué función deberías usar para eso? ¡Ya lo sabes!

command-line

```
>>> len(lottery)
6
```

¡Sí! `len()` puede darte el número de objetos en una lista. Útil, ¿verdad? Tal vez la ordenemos ahora:

command-line

```
>>> lottery.sort()
```

No devuelve nada, sólo ha cambiado el orden en que los números aparecen en la lista. Vamos a imprimirla otra vez y ver que ha pasado:

command-line

```
>>> print(lottery)
[3, 12, 19, 30, 42, 59]
```

Como puedes ver, los números de tu lista ahora están ordenados de menor a mayor. ¡Enhorabuena!

¿Te gustaría invertir ese orden? ¡Vamos a hacerlo!

command-line

```
>>> lottery.reverse()
>>> print(lottery)
[59, 42, 30, 19, 12, 3]
```

Si quieres añadir algo a tu lista, puedes hacerlo escribiendo este comando:

command-line

```
>>> lottery.append(199)
>>> print(lottery)
[59, 42, 30, 19, 12, 3, 199]
```

Si deseas mostrar sólo el primer número, puedes hacerlo mediante el uso de **indexes** (en español, índices). Un índice es el número que te dice dónde en una lista aparece un ítem. Las programadoras y los programadores prefieren comenzar a contar desde 0, por lo tanto el primer objeto en tu lista esta en el indice 0, el próximo esta en el 1, y así sucesivamente. Intenta esto:

command-line

```
>>> print(lottery[0])
59
>>> print(lottery[1])
42
```

Como puedes ver, puedes acceder a diferentes objetos en tu lista utilizando el nombre de la lista y el índice del objeto dentro de corchetes.

Para borrar algo de tu lista tendrás que usar **índices** como aprendimos anteriormente y la función `pop()`. Vamos a tratar de ejemplificar esto y reforzaar lo que aprendimos anteriormente; vamos a borrar el primer número de nuestra lista.

command-line

```
>>> print(lottery)
[59, 42, 30, 19, 12, 3, 199]
>>> print(lottery[0])
59
>>> lottery.pop(0)
59
>>> print(lottery)
[42, 30, 19, 12, 3, 199]
```

Funcionó de maravilla!

Para diversión adicional, prueba algunos otros índices: 6, 7, 1000, -1, -6 ó -1000. A ver si se puedes predecir el resultado antes de intentar el comando. ¿Tienen sentido los resultados?

Puedes encontrar una lista de todos los métodos disponibles para listas en este capítulo de la documentación de Python: https://docs.python.org/3/tutorial/datastructures.html

Diccionarios

> Para lectores en casa: este capítulo está cubierto en el video Bases de Python: Diccionarios.

Un diccionario es similar a una lista, pero accedes a valores usando una llave en vez de un índice. Una llave puede ser cualquier cadena o número. La sintaxis para definir un diccionario vacío es:

command-line

```
>>> {}
{}
```

Esto demuestra que acabas de crear un diccionario vacío. ¡Hurra!

Ahora, trata escribiendo el siguiente comando (intenta reemplazando con tu propia información):

command-line

```
>>> participant = {'name': 'Ola', 'country': 'Poland', 'favorite_numbers': [7, 42, 92]}
```

Con este comando, acabas de crear una variable llamada `participant` con tres pares llave-valor:

- La llave `name` apunta al valor `'Ola'` (un objeto `string`),
- `country` apunta a `'Poland'` (otro `string`),
- y `favorite_numbers` apunta a `[7, 42, 92]` (una `list` con tres números en ella).

Puedes verificar el contenido de claves individuales con esta sintaxis:

command-line

```
>>> print(participant['name'])
Ola
```

Lo ves, es similar a una lista. Pero no necesitas recordar el índice - sólo el nombre.

¿Qué pasa si le pedimos a Python el valor de una clave que no existe? ¿Puedes adivinar? ¡Pruébalo y verás!

command-line

```
>>> participant['age']
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
KeyError: 'age'
```

¡Mira, otro error! Este es un **KeyError**. Python te ayuda y te dice que la llave `'age'` no existe en este diccionario.

¿Cuando deberías usar un diccionario o una lista? Bueno, es un buen punto para reflexionar. Piensa sobre la respuesta, antes de mirar una solución en la siguiente línea.

- ¿Sólo necesitas una secuencia ordenada de elementos? Usa una lista.
- ¿Necesitas asociar valores con claves, así puedes buscarlos eficientemente (usando las claves) más adelante? Utiliza un diccionario.

Los diccionarios, como las listas, son *mutables*, lo que quiere decir que pueden ser modificados después de ser creados. Puedes agregar nuevos pares llave/valor a un diccionario luego de crearlo, como:

command-line

```
>>> participant['favorite_language'] = 'Python'
```

Como las listas, usando el método `len()` en los diccionarios devulven el número de pares llave-valor en el diccionario. Adelante escribe el comando:

command-line

```
>>> len(participant)
4
```

Espero tenga sentido hasta ahora. :) ¿Listo para más diversión con diccionarios? Salta a la siguiente línea para algunas cosas sorprendentes.

Puedes utilizar el comando `pop()` para borrar un elemento en el diccionario. Por ejemplo, si deseas eliminar la entrada correspondiente a la clave `'favorite_numbers'`, tienes que escribir el siguiente comando:

command-line

```
>>> participant.pop('favorite_numbers')
[7, 42, 92]
>>> participant
{'country': 'Poland', 'favorite_language': 'Python', 'name': 'Ola'}
```

Como puedes ver en la salida, el par de llave-valor correspondiente a la llave 'favorite_numbers' ha sido eliminado.

Además de esto, también puedes cambiar un valor asociado a una llave ya creada en el diccionario. Escribe:

command-line

```
>>> participant['country'] = 'Germany'
>>> participant
{'country': 'Germany', 'favorite_language': 'Python', 'name': 'Ola'}
```

Como puedes ver, el valor de la llave `'country'` ha sido modificado de `'Poland'` a `'Germany'`. :) ¿Emocionante? ¡Hurra! Has aprendido otra cosa asombrosa.

Resumen

¡Genial! Sabes mucho sobre programación ahora. En esta última parte aprendiste sobre:

- **errores** - ahora sabes cómo leer y entender los errores que aparecen si Python no entiende un comando
- **variables** - nombres para los objetos que te permiten codificar más fácilmente y hacer el código más legible
- **listas** - listas de objetos almacenados en un orden determinado
- **diccionarios** - objetos almacenados como pares llave-valor

¿Emocionada por la siguiente parte? :)

Compara cosas

> Para lectores en casa: este capítulo está cubierto en el video Bases de Python: Comparaciones.

Buena parte de la programación incluye comparar cosas. ¿Qué es lo más fácil para comparar? Números, por supuesto. Vamos a ver cómo funciona:

command-line

```
>>> 5 > 2
True
>>> 3 < 1
False
>>> 5 > 2 * 2
True
>>> 1 == 1
True
>>> 5 != 2
True
```

Le dimos a Python algunos números para comparar. Como puedes ver, Python no sólo puede comparar números, sino que también puede comparar resultados de funciones. Bien, ¿eh?

¿Te preguntas por qué pusimos dos signos igual `==` al lado del otro para comparar si los números son iguales? Utilizamos un solo `=` para asignar valores a las variables. Siempre, **siempre** es necesario poner dos `==` Si deseas comprobar que las cosas son iguales entre sí. También podemos afirmar que las cosas no son iguales a otras. Para eso, utilizamos el símbolo `!=`, como mostramos en el ejemplo anterior.

Da dos tareas más a Python:

command-line

```
>>> 6 >= 12 / 2
True
>>> 3 <= 2
False
```

Hemos visto `>` y `<`, pero, ¿qué significan `>=` y `<=`? Los puedes leer así:

- x `>` y significa: x es mayor que y
- x `<` y significa: x es menor que y
- x `<=` y significa: x es menor o igual que y
- x `>=` y significa: x es mayor o igual que y

¡Genial! ¿Quieres hacer uno mas? Intenta esto:

command-line

```
>>> 6 > 2 and 2 < 3
True
>>> 3 > 2 and 2 < 1
False
>>> 3 > 2 or 2 < 1
True
```

Puedes darle a Python todos los números para comparar que quieras, y siempre te dará una respuesta. Muy inteligente, ¿verdad?

- **and** - si utilizas el operador `and`, ambas comparaciones deben ser True para que el resultado de todo el comando sea True
- **or** - si utilizas el operador `or`, sólo una de las comparaciones tiene que ser True para que el resultado de todo el comando sea True

¿Has oído la expresión "comparar manzanas con naranjas"? Vamos a probar el equivalente en Python:

command-line

```
>>> 1 > 'django'
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: '>' not supported between instances of 'int' and 'str'
```

Aquí verás que al igual que en la expresión, Python no es capaz de comparar un número (`int`) y un string (`str`). En cambio, muestra un **TypeError** y nos dice que los dos tipos no se pueden comparar.

Boolean

Incidentalmente, acabas de aprender sobre un nuevo tipo de objeto en Python. Se llama **Boolean** (booleano).

Hay sólo dos objetos booleanos:

- True - verdadero
- False - falso

Pero para que Python entienda esto, siempre los tienes que escribir de modo 'True' (la primera letra en mayúscula, con el resto de las letras en minúscula). **true, TRUE, y tRUE no funcionarán – solamente True es correcta.** (Lo mismo aplica también para 'False'.)

Los valores booleanos pueden ser variables, también. Ve el siguiente ejemplo:

command-line

```
>>> a = True
>>> a
True
```

También puedes hacerlo de esta manera:

command-line

```
>>> a = 2 > 5
>>> a
False
```

Practica y diviértete con los booleanos ejecutando los siguientes comandos:

- `True and True`
- `False and True`
- `True or 1 == 1`
- `1 != 2`

¡Felicidades! Los booleanos son una de las funciones más geniales en programación y acabas de aprender cómo usarlos.

¡Guárdalo!

> Para lectores en casa: este capítulo está cubierto en el video Bases de Python: Guardando archivos y condicionales.

Hasta ahora hemos escrito todo nuestro código Python en el intérprete, lo cual nos limita a ingresar una línea de código a la vez. Normalmente los programas son guardados en archivos y son ejecutados por el **intérprete** o **compilador** de nuestro lenguaje de programación. Hasta ahora, hemos estado corriendo nuestros programas de a una línea por vez en el **intérprete** de Python. Necesitaremos más de una línea de código para las siguientes tareas, entonces necesitaremos hacer rápidamente lo que sigue:

- Salir del intérprete de Python
- Abrir el editor de texto de nuestra elección
- Guardar algo de código en un nuevo archivo de Python
- ¡Ejecutarlo!

Para salir del intérprete de Python que hemos estado usando, escribe `exit()`

command-line

```
>>> exit()
$
```

Esto te llevará de vuelta a la línea de comandos.

Anteriormente, elegimos un editor de código en la sección de Editor de código. Tendremos que abrir el editor ahora y escribir algo de código en un archivo nuevo:

editor

```
print('Hello, Django girls!')
```

Obviamente, ahora eres una desarrolladora Python muy experimentada, asi que sintete libre de escribir algo del código que has aprendido hoy.

Ahora tenemos que guardar el archivo y asignarle un nombre descriptivo. Vamos a llamar al archivo **python_intro.py** y guardarlo en tu escritorio. Podemos nombrar el archivo como queramos, pero la parte importante es asegurarse de que termina en **.py**. La extensión **.py** le dice a nuestro sistema operativo que es un **archivo ejecutable de python** y Python lo puede ejecutar.

> **Nota** Deberías notar una de las cosas más geniales de los editores de código: ¡los colores! En la consola de Python, todo era del mismo color, ahora deberías ver que la función `print` es de un color diferente a la cadena en su interior. Esto de denomina "sintaxis resaltada", y es una característica muy útil cuando se programa. El color de las cosas te dará pistas, como cadenas no cerradas o errores tipográficos en un nombre clave (como `def` en una función, que veremos a continuación). Esta es una de las razones por las cuales usar un editor de código. :)

Con el archivo guardado, ¡es hora de ejecutarlo! Utilizando las habilidades que has aprendido en la sección de línea de comandos, utiliza la terminal para **cambiar los directorios** e ir al escritorio.

Change directory: OS X
En una Mac, el comando se verá algo como esto:

command-line

```
$ cd ~/Desktop
```

Change directory: Linux
En Linux, va a ser así (la palabra "Desktop" puede estar traducida a tu idioma, escritorio en español):

command-line

```
$ cd ~/Desktop
```

Change directory: Windows Command Prompt
En Command Prompt en windows, sera así:

command-line

```
> cd %HomePath%\Desktop
```

Change directory: Windows Powershell
Y en Powershell, será así:

command-line

```
> cd $Home\Desktop
```

Si te atoras, pide ayuda. ¡Justo para eso están aquí los asesores!

Ahora usa Python para ejecutar el código en el archivo así:

command-line

```
$ python3 python_intro.py
Hello, Django girls!
```

Nota: En Windows 'python3' no es reconocido como comando. En vez usa 'python' para ejecutar el archivo:

command-line

```
> python python_intro.py
```

¡Muy bien! Ejecutaste tu primer programa de Python desde un archivo. ¿Te sientes increíble?

Ahora puedes moverte a una herramienta esencial en la programación:

If … elif … else

Un montón de cosas en el código sólo deberían ser ejecutadas cuando se cumplan ciertas condiciones. Por eso Python tiene algo llamado **sentencias if**.

Reemplaza el código en tu archivo **python_intro.py** con esto:

python_intro.py

```
if 3 > 2:
```

Si lo guardáramos y lo ejecutáramos, veríamos un error como este:

command-line

```
$ python3 python_intro.py
File "python_intro.py", line 2
         ^
SyntaxError: unexpected EOF while parsing
```

Python espera que le demos más instrucciones las cuales se ejecutan si la condición `3 > 2` es verdadera (o `True`). Intentemos hacer que Python imprima "It works!". Cambia tu código en el archivo **python_intro.py** para que se vea como esto:

python_intro.py

```
if 3 > 2:
    print('It works!')
```

¿Observas cómo hemos indentado la siguiente línea de código con 4 espacios? Necesitamos hacer esto para que Python sepa que código ejecutar cuando la condición es verdadera. Podrías poner un solo espacio, pero casi todas las programadores y programadores de Python ponen 4 espacios para que el código sea más legible. Un tabulador también cuenta como 4 espacios (dependiendo de la configuración de tu editor). Elige una forma para indentar, ¡y hazlo siempre igual! Si ya has indentado el código con 4 espacios, hazlo siempre igual. Es importante que la indentación del código sea consistente para evitar problemas.

Guárdalo y ejecútalo de nuevo:

command-line

```
$ python3 python_intro.py
It works!
```

Nota: Recuerda que en Windows, 'python3' no es reconocido como comando. De ahora en adelante, reemplaza 'python3' con 'python' para ejecutar el archivo.

¿Qué pasa si una condición no es verdadera?

En ejemplos anteriores, el código fue ejecutado sólo cuando las condiciones eran ciertas. Pero Python también tiene declaraciones `elif` y `else`:

python_intro.py

```
if 5 > 2:
    print('5 is indeed greater than 2')
else:
    print('5 is not greater than 2')
```

Cuando esto se ejecute imprimirá:

command-line

```
$ python3 python_intro.py
5 is indeed greater than 2
```

Si 2 fuera mayor que 5, entonces se ejecutaría el segundo comando. Vamos a ver como funciona `elif`:

python_intro.py

```
name = 'Sonja'
if name == 'Ola':
    print('Hey Ola!')
elif name == 'Sonja':
    print('Hey Sonja!')
else:
    print('Hey anonymous!')
```

y al ejecutarlo:

command-line

```
$ python3 python_intro.py
Hey Sonja!
```

¿Ves lo que pasó allí? `elif` te permite agregar condiciones adicionales que se ejecutan si la condición previa falla.

Puede agregar tantas sentencias `elif` como quieras despues de la sentencia `if` inicial. Por ejemplo:

python_intro.py

```
volume = 57
if volume < 20:
    print("It's kinda quiet.")
elif 20 <= volume < 40:
    print("It's nice for background music")
elif 40 <= volume < 60:
    print("Perfect, I can hear all the details")
elif 60 <= volume < 80:
    print("Nice for parties")
elif 80 <= volume < 100:
    print("A bit loud!")
else:
    print("Me duelen las orejas! :(")
```

Python corre a través de cada prueba en secuencia e imprime:

command-line

```
$ python3 python_intro.py
Perfect, I can hear all the details
```

Comentarios

Las líneas que empiezan por `#` son comentarios. Puedes escribir lo que quieres detrás del `#` y Python lo ignorará. Los comentarios sirven para explicar el código a la gente que lo lea en el futuro (¡o incluso a ti misma!).

A ver qué pinta tiene:

python_intro.py

```
# Cambiar el volumen si esta muy alto o muy bajo
if volume < 20 or volume > 80:
    volume = 50
    print("Mucho mejor!")
```

No es necesario que escribas un comentario por cada linea de código, pero son muy útiles para explicar por qué tu código hace algo, o para resumir cuando esta haciendo algo complejo.

Resumen

En los últimos tres ejercicios aprendiste acerca de:

- **Comparar cosas** - en Python puedes comparar cosas haciendo uso de `>`, `>=`, `==`, `<=`, `<` y de los operatores `and` y `or`
- **Boolean** - un tipo de objeto que sólo puede tener uno de dos valores: `True` o `False`
- **Guardar archivos** - almacenar código en archivos para que puedas ejecutar programas más grandes.
- **if... elif... else** - sentencias que te permiten ejecutar código sólo cuando se cumplen ciertas condiciones.
- **comentarios** - líneas que Python no ejecutará que permiten documentar el código

¡Es hora de leer la última parte de este capítulo!

¡Tus propias funciones!

> Para lectores en casa: este capítulo está cubierto en el video Bases de Python: Funciones.

¿Recuerdas funciones como `len()` que puedes ejecutar en Python? Bueno, te tenemos buenas noticias, ¡ahora aprenderás a escribir tus propias funciones!

Una función es una secuencia de instrucciones que Python debe ejecutar. Cada función en Python comienza con la palabra clave `def`, se le asigna un nombre y puede tener algunos parámetros. Vamos a probar. Reemplaza el código en **python_intro.py** con lo siguiente:

python_intro.py

```
def hi():
    print('Hi there!')
    print('How are you?')

hi()
```

Bien, ¡nuestra primera función está lista!

Te preguntarás por qué hemos escrito el nombre de la función en la parte inferior del archivo. Esto es porque Python lee el archivo y lo ejecuta desde arriba hacia abajo. Así que para poder utilizar nuestra función, tenemos que reescribir su nombre en la parte inferior.

Ejecutemos esto y veamos qué sucede:

command-line

```
$ python3 python_intro.py
Hi there!
How are you?
```

Nota: Si no funcionó, no te preocupes! La salida le ayudará a entender por qué:

- Si te sale `NameError`, probablemente significa que escribiste algo mal, así que deberias comprobar si utilizaste el mismo nombre para crear la función con `def hi():` y al llamarla con `hi()`.
- Si te sale un `IndentationError`, comprueba que las líneas del `print` tienen el mismo espacio en blanco al comienzo de línea: python requiere que todo el código dentro de la función esté perfectamente alineado.
- Si no hay ninguna salida, comprueba que el último `hi()` *no esté* identado - si lo está, esa línea también sera parte de la función, y nunca se ejecutará.

Escribamos nuestra primera función con parametros. Usaremos el ejemplo anterior - una funcion que dice 'hi' a la persona ejecutandola - con un nombre:

python_intro.py

```
def hi(name):
```

Como puedes ver, ahora dimos a nuestra función un parámetro que llamamos `name`:

python_intro.py

```
def hi(name):
    if name == 'Ola':
        print('Hi Ola!')
    elif name == 'Sonja':
        print('Hi Sonja!')
    else:
        print('Hi anonymous!')

hi()
```

Recuerda: La función `print` está indentada cuatro espacios dentro del `if`. Esto es porque la función corre cuando la condición se cumple. Veremos como funciona ahora:

command-line

```
$ python3 python_intro.py
Traceback (most recent call last):
File "python_intro.py", line 10, in <module>
  hi()
TypeError: hi() missing 1 required positional argument: 'name'
```

Oops, un error. Por suerte, Python nos da un mensaje de error bastante útil. Nos dice que la función `hi()` (la que definimos) tiene un argumento requerido (llamado `name`) y que se nos olvidó pasarlo al llamar a la función. Vamos a arreglarlo en la parte inferior del archivo:

python_intro.py

```
hi("Ola")
```

Y lo ejecutamos de nuevo:

command-line

```
$ python3 python_intro.py
Hi Ola!
```

¿Y si cambiamos el nombre?

python_intro.py

```
hi("Sonja")
```

Y lo ejecutamos:

command-line

```
$ python3 python_intro.py
Hi Sonja!
```

Ahora, ¿qué crees que suceda si escribes otro nombre ahí? (No Ola ni Sonja) Inténtalo y ve si tienes razón. Debería imprimir esto:

command-line

```
Hi anonymous!
```

Esto es increíble, ¿verdad? De esta forma no tienes que repetir todo cada vez que deseas cambiar el nombre de la persona a la que la función debería saludar. Y esa es exactamente la razón por la que necesitamos funciones - ¡para no repetir tu código!

Vamos a hacer algo más inteligente - hay más de dos nombres, y escribir una condición para cada uno sería difícil, ¿no?

python_intro.py

```
def hi(name):
    print('Hi ' + name + '!')

hi("Rachel")
```

Ahora vamos a llamar al código:

command-line

```
$ python3 python_intro.py
Hi Rachel!
```

¡Felicidades! Acabas de aprender cómo escribir funciones :)

Bucles

> Para lectores en casa: este capítulo está cubierto en el video Python Basics: For Loop.

Esta es la última parte. ¡Qué rápido! ¿no? :)

A las programadoras no les gusta repetirse a si mismas. Programar es sobre automatizar cosas, entonces no queremos saludar a cada persona por su nombre manualmente, ¿no? Ahí es donde los loops son útiles.

¿Todavía recuerdas las listas? Hagamos una lista de las chicas:

python_intro.py

```
girls = ['Rachel', 'Monica', 'Phoebe', 'Ola', 'You']
```

Queremos saludar a todas ellas por su nombre. Tenemos la función `hi` que hace eso, así que vamos a usarla en un bucle:

python_intro.py

```
for name in girls:
```

La sentencia `for` funciona de forma parecida a la sentencia `if`; el código de dentro tiene que estar indentado con cuatro espacios en ambas.

Aquí está el código completo que estará en el archivo:

python_intro.py

```
def hi(name):
    print('Hi ' + name + '!')

girls = ['Rachel', 'Monica', 'Phoebe', 'Ola', 'You']
for name in girls:
    hi(name)
    print('Next girl')
```

Y cuando lo ejecutamos:

command-line

```
$ python3 python_intro.py
Hi Rachel!
Next girl
Hi Monica!
Next girl
Hi Phoebe!
Next girl
Hi Ola!
Next girl
Hi You!
Next girl
```

Como puedes ver, todo lo que pones con una indentación dentro de una sentencia `for` será repetido para cada elemento de la lista `girls`.

También se puede usar `for` con números, usando la función `range` :

python_intro.py

```
for i in range(1, 6):
    print(i)
```

Lo que imprimirá:

command-line

```
1
2
3
4
5
```

`range` es una función que crea una lista de números en serie (estos números son proporcionados por ti como parámetros).

Ten en cuenta que el segundo de estos dos números no será incluido en la lista que retornará Python (es decir, `range(1, 6)` cuenta desde 1 a 5, pero no incluye el número 6). Eso pasa porque "range" está medio-abierto, y con eso queremos decir que incluye el primer valor, pero no el último.

Resumen

Eso es todo. **¡Eres genial!** Este fue un capítulo difícil, por lo que debes sentirte orgullosa de ti misma. ¡Nosotras estamos orgullosas de ti porque has llegado lejos!

Si quieres ver un tutorial de Python oficial y completo, puedes visitar https://docs.python.org/3/tutorial/. Este tutorial te enseñará conocimientos más completos y detallados del lenguaje. ¡Bravo! :)

Tal vez quieras hacer algo distinto por un momento - estirarte, caminar un poco, descansar tus ojos - antes de pasar al siguiente capítulo. :)

¿Qué es Django?

Django (*gdh/ˈdʒæŋgoʊ/jang-goh*) es un framework de aplicaciones web gratuito y de código abierto (open source) escrito en Python. Un framework web es un conjunto de componentes que te ayudan a desarrollar sitios web más fácil y rápidamente.

Cuando construyes un sitio web, siempre necesitas un conjunto de componentes similares: una manera de manejar la autenticación de usuarios (registrarse, iniciar sesión, cerrar sesión), un panel de administración para tu sitio web, formularios, una forma de subir archivos, etc.

Por suerte para nosotros, hace tiempo que otros desarrolladores se dieron cuenta de que siempre se enfrentaban a los mismos problemas cuando construían sitios web, y por eso se unieron y crearon frameworks (Django es uno de ellos) con componentes listos para usarse.

Los frameworks sirven para que no tengamos que reinventar la rueda cada vez y que podamos avanzar más rápido al construir un nuevo sitio.

¿Por qué necesitas un framework?

Para entender para que sirve realmente Django, necesitamos fijarnos en cómo funcionan los servidores. Lo primero es que el servidor necesita enterarse de que tu quieres que te sirva una página web.

Imagina un buzón (puerto) en el que alguien está constantemente mirando si hay cartas entrantes (peticiones). Esto es lo que hace un servidor web. El servidor web lee la carta, y envía una respuesta con la página web. Pero para enviar algo, tenemos que tener algún contenido. Y Django nos ayuda a crear ese contenido.

¿Qué sucede cuando alguien solicita una página web de tu servidor?

Cuando llega una petición a un servidor web, es pasada a Django quien intenta averiguar que es realmente solicitado. Toma primero una dirección de página web e intenta averiguar qué hacer. Esta parte es realizada por el **urlresolver** de Django (ten en cuenta que la dirección de un sitio web es llamada URL - Uniform Resource Locator; así que el nombre *urlresolver* tiene sentido). Este no es muy inteligente - toma una lista de patrones y trata de hacer coincidir la URL. Django comprueba los patrones de arriba hacia abajo y si algo coincide entonces Django le pasa la solicitud a la función asociada (que se llama *view (vista)*).

Imagina a un cartero llevando una carta. Él está caminando por la calle y comprueba cada número de casa con el que está en la carta. Si coincide, deja la carta allí. Así es como funciona el urlresolver!

En la función de *view (vista)* se hacen todas las cosas interesantes: podemos mirar a una base de datos para buscar alguna información. ¿Tal vez el usuario pidió cambiar algo en los datos? Como una carta diciendo "Por favor cambia la descripción de mi trabajo." La *vista* puede comprobar si tienes permiso para hacerlo, actualizar la descripción de tu trabajo y devolver un mensaje: "¡Hecho!". Luego la *vista* genera una respuesta y Django puede enviarla al navegador del usuario.

Esta descripción es un poco simplista, pero de momento no necesitas saber todos los detalles técnicos, con tener una idea general es más que suficiente.

Así que en lugar de detenernos demasiado en los detalles, vamos a empezar a crear algo con Django y ¡así aprenderemos las cosas importantes sobre la marcha!

Instalacion de Django

> **Nota** Si usas un Chromebook, omite este capítulo y asegúrate de seguir las instrucciones de Chromebook Setup.
>
> **Nota** Si ya has realizado los pasos de instalación, esto ya lo has hecho. ¡Puedes avanzar directamente al siguiente capítulo!
>
> Parte de esta sección se basa en tutoriales por Geek Girls Carrots (https://github.com/ggcarrots/django-carrots).
>
> Parte de este capítulo está basada en el django-marcador tutorial bajo la licencia Creative Commons Attribution-ShareAlike 4.0 internacional. El tutorial de django-marcador tiene derechos de autor de Markus Zapke-Gündemann et al.

Entorno virtual

Antes de instalar Django, instalaremos una herramienta extremadamente útil que ayudará a mantener tu entorno de desarrollo ordenado en tu computadora. Es posible saltarse este paso, pero es altamente recomendable. ¡Empezar con la mejor configuración posible te ahorrará muchos problemas en el futuro!

Así que, vamos a crear un **entorno virtual** (también llamado un *virtualenv*). Virtualenv aísla tu configuración de Python/Django por cada proyecto. Esto quiere decir que cualquier cambio que hagas en un sitio web no afectará a ningún otro que estés desarrollando. Genial, ¿no?

Todo lo que necesitas hacer es encontrar un directorio en el que quieras crear el `virtualenv`; tu directorio home, por ejemplo. En Windows, puede verse como `C:\Users\Name` (donde `Name` es el nombre de tu usuario).

> **NOTA:** En Windows, asegúrate de que este directorio no contiene caracteres especiales o acentuados; si tu nombre de usuario contiene caracteres acentuados, usa un directorio distinto, por ejemplo `C:\djangogirls`.

Para este tutorial usaremos un nuevo directorio `djangogirls` en tu directorio home:

command-line

```
$ mkdir djangogirls
$ cd djangogirls
```

Haremos un virtualenv llamado `myvenv`. El comando general estará en el formato:

command-line

```
$ python3 -m venv myvenv
```

Virtual environment: Windows
Para crear un nuevo `virtualenv`, necesitas abrir una terminal "command prompt" y ejecutar `python -m venv myvenv`. Se verá así:

command-line

```
C:\Users\Name\djangogirls> python -m venv myvenv
```

Donde `myvenv` es el nombre de tu `virtualenv`. Puedes utilizar cualquier otro nombre, pero asegúrate de usar minúsculas y no usar espacios, acentos o caracteres especiales. También es una buena idea mantener el nombre corto. ¡Vas utilizarlo muchas veces!

Virtual environment: Linux and OS X
Podemos crear un `virtualenv` en Linux y OS X, es tan sencillo como ejecutar `python3 -m venv myvenv`. Se verá así:

command-line

```
$ python3 -m venv myvenv
```

`myvenv` es el nombre de tu `virtualenv`. Puedes usar cualquier otro nombre, pero sólo utiliza minúsculas y no incluyas espacios. También es una buena idea mantener el nombre corto. ¡Vas a referirte muchas veces a él!

> **NOTA:** En algunas versiones de Debian/Ubuntu, puede que obtengas el siguiente error:
>
> command-line
>
> ```
> The virtual environment was not created successfully because ensurepip is not available.  En sistemas Debian/U
> buntu, tendrás que instalar el paquete python3-venv usando el siguiente comando.
>  apt-get install python3-venv
> Puede que tengas que usar sudo con este comando.  Después de instalar el paquete python3-venv, vuelve a crear
> tu entorno virtual.
> ```
>
> En este caso, sigue las instrucciones anteriores e instala el paquete `python3-venv`:
>
> command-line
>
> ```
> $ sudo apt install python3-venv
> ```
>
> **NOTA:** En algunas versiones de Debian/Ubuntu inicializar el entorno virtual de esta manera da el siguiente error:
>
> command-line
>
> ```
> Error: Command '['/home/eddie/Slask/tmp/venv/bin/python3', '-Im', 'ensurepip', '--upgrade', '--default-pip']'
> returned non-zero exit status 1
> ```
>
> Para evitar esto, utiliza directamente el comando `virtualenv`.
>
> command-line
>
> ```
> $ sudo apt-get install python-virtualenv
> $ virtualenv --python=python3.6 myvenv
> ```
>
> **NOTA:** Si obtienes un error como
>
> command-line
>
> ```
> E: Unable to locate package python3-venv
> ```
>
> entonces ejecuta:
>
> command-line
>
> ```
> sudo apt install python3.6-venv
> ```

Trabajar con virtualenv

El comando anterior creará un directorio llamado `myvenv` (o cualquier nombre que hayas elegido) que contiene nuestro entorno virtual (básicamente un montón de archivos y carpetas).

Working with virtualenv: Windows
Inicia el entorno virtual ejecutando:

command-line

```
C:\Users\Name\djangogirls> myvenv\Scripts\activate
```

> **Nota:** en 10 de Windows puedes obtener un error en Windows PowerShell que dice `execution of scripts is disabled on this system`. En este caso, abre otro Windows PowerShell con la opción "Ejecutar como administrador". Luego intenta escribir el siguiente comando antes de inicializar tu entorno virtual:
>
> command-line
>
> ```
> C:\WINDOWS\system32 > Set-ExecutionPolicy -ExecutionPolicy RemoteSigned
>  Execution Policy Change
>  The execution policy helps protect you from scripts that you do not trust. Changing the execution policy mi
> ght expose you to the security risks described in the about_Execution_Policies help topic at http://go.microso
> ft.com/fwlink/?LinkID=135170. Do you want to change the execution policy? [Y] Yes  [A] Yes to All  [N] No  [L]
>  No to All  [S] Suspend  [?] Help (default is "N"): A
> ```

Working with virtualenv: Linux and OS X
Inicia el entorno virtual ejecutando:

command-line

```
$ source myvenv/bin/activate
```

¡Recuerda reemplazar `myvenv` con tu nombre de `virtualenv` que hayas elegido!

> **NOTA:** a veces `source` podría no estar disponible. En ese caso trata hacerlo de esta forma:
>
> command-line
>
> ```
> $ . myvenv/bin/activate
> ```

Sabrás que tienes `virtualenv` iniciado cuando veas que la línea de comando en tu consola tiene el prefijo `(myvenv)`.

Cuando trabajes en un entorno virtual, `python` automáticamente se referirá a la versión correcta, de modo que puedes utilizar `python` en vez de `python3`.

Ok, tenemos todas las dependencias importantes en su lugar. ¡Finalmente podemos instalar Django!

Instalar Django

Ahora que tienes tu `virtualenv` iniciado, puedes instalar Django.

Antes de hacer eso, debemos asegurarnos que tenemos la última versión de `pip`, el software que utilizamos para instalar Django:

command-line

```
(myvenv) ~$ python3 -m pip install --upgrade pip
```

Instalar paquetes con un fichero de requisitos (requirements)

Un fichero de requisitos (requirements) tiene una lista de dependencias que se deben instalar mediante `pip install`:

Lo primero, crea un fichero `requirements.txt` dentro del directorio `djangogirls/`, usando el editor de código que hemos instalado hace un momento:

```
djangogirls
└──requirements.txt
```

Dentro del fichero `djangogirls/requirements.txt` deberías tener el siguiente texto:

djangogirls/requirements.txt

```
Django~=2.0.6
```

Ahora, ejecuta `pip install -r requirements.txt` para instalar Django.

command-line

```
(myvenv) ~$ pip install -r requirements.txt
Collecting Django~=2.0.6 (from -r requirements.txt (line 1))
  Downloading Django-2.0.6-py3-none-any.whl (7.1MB)
Installing collected packages: Django
Successfully installed Django-2.0.6
```

Installing Django: Windows

> Si obtienes un error al ejecutar pip en Windows comprueba si la ruta de tu proyecto contiene espacios, acentos o caracteres especiales (por ejemplo, `C:\Users\User Name\djangogirls`). Si los tiene, por favor considera moverla a otro lugar sin espacios, acentos o caracteres especiales (sugerencia: `C:\djangogirls`). Crea un nuevo virtualenv en el nuevo directorio, luego borra el viejo y reintenta el comando anterior. (Mover el directorio de virtualenv no funciona puesto que virtualenv usa rutas absolutas.)

Installing Django: Windows 8 and Windows 10

> Puede que tu línea de comandos se congele después de que intentes instalar Django. Si esto sucede, en vez del comando anterior utiliza:
>
> command-line
>
> ```
> C:\Users\Name\djangogirls> python -m pip install -r requirements.txt
> ```

Installing Django: Linux

> Si obtienes un error al ejecutar pip en Ubuntu 12.04 ejecuta `python -m pip install -U --force-reinstall pip` para arreglar la instalación de pip en el virtualenv.

¡Eso es todo! Ahora estás lista (por fin) para crear una aplicación Django!

¡Tu primer proyecto en Django!

> Parte de este capítulo se basa en tutoriales por Geek Girls Carrots (https://github.com/ggcarrots/django-carrots).
>
> Parte de este capítulo está basado en el tutorial django-marcador bajo licencia de Creative Commons Attribution-ShareAlike 4.0 internacional. El tutorial de django-marcador tiene derechos de autor de Markus Zapke-Gündemann et al.

¡Vamos a crear un blog sencillo!

El primer paso es iniciar un nuevo proyecto de Django. Básicamente, significa que vamos a lanzar unos scripts proporcionados por Django que nos crearán el esqueleto de un proyecto de Django. Son solo un montón de directorios y archivos que usaremos más tarde.

Los nombres de algunos archivos y directorios son muy importantes para Django. No deberías renombrar los archivos que estamos a punto de crear. Moverlos a un lugar diferente tampoco es buena idea. Django necesita mantener una cierta estructura para poder encontrar cosas importantes.

> Recuerda ejecutar todo en el virtualenv. Si no ves un prefijo `(myvenv)` en tu consola tienes que activar tu virtualenv. Explicamos cómo hacerlo en el capítulo de **Instalación de Django** en la sección **Trabajar con virtualenv**. Basta con escribir `myvenv\Scripts\activate` en Windows o `source myvenv/bin/activate` en Mac OS / Linux.

Create project: OS X or Linux
En MacOS o Linux deberías ejecutar el siguiente comando en la consola. **no te olvides de añadir el punto `.` al final**

command-line

```
(myvenv) ~/djangogirls$ django-admin startproject mysite .
```

> El punto `.` es crucial porque le dice al script que instale Django en el directorio actual (para el cual el punto `.` sirve de abreviatura).
>
> **Nota** Cuando escribas los comandos de arriba acuérdate de que sólo tienes que escribir la parte que empieza por `django-admin`. La parte de `(myvenv) ~/djangogirls$` que mostramos aquí es sólo un ejemplo del mensaje que aparecerá en tu línea de comandos.

Create project: Windows
En Windows debes ejecutar el siguiente comando. **(No olvides incluir el punto `.` al final)**:

command-line

```
(myvenv) C:\Users\Name\djangogirls> django-admin.exe startproject mysite .
```

> El punto `.` es crucial porque le dice al script que instale Django en el directorio actual (para el cual el punto `.` sirve de abreviatura).
>
> **Nota** Cuando teclees los comandos de arriba, recuerda que sólo tienes que escribir la parte que empieza por `django-admin.exe`. La parte de `(myvenv) C:\Users\Name\djangogirls>` que mostramos aquí es sólo un ejemplo del mensaje que aparecerá en tu línea de comandos.

`django-admin.py` es un script que creará los archivos y directorios para ti. Ahora deberías tener una estructura de directorios parecida a esta:

```
djangogirls
├───manage.py
├───mysite
│        settings.py
│        urls.py
│        wsgi.py
│        __init__.py
└───requirements.txt
```

> **Nota**: en tu estructura de directorios, tambièn veràs el directorio `venv` que creamos anteriormente.

`manage.py` es un script que ayuda con la administración del sitio. Con él podremos iniciar un servidor web en nuestro ordenador sin necesidad de instalar nada más, entre otras cosas.

El archivo `settings.py` contiene la configuración de tu sitio web.

Recuerdas cuando hablamos de una cartera que debía comprobar dónde entregar una carta? El archivo `urls.py` contiene una lista de los patrones utilizados por `urlresolver`.

Por ahora vamos a ignorar el resto de archivos porque no los vamos a cambiar. ¡Sólo acuérdate de no borrarlos accidentalmente!

Cambiar la configuración

Vamos a hacer algunos cambios en `mysite/settings.py`. Abre el archivo usando el editor de código que has instalado anteriormente.

Nota: Ten en cuenta que `settings.py` es un archivo normal, como cualquier otro. Puedes abrirlo con el editor de texto, usando "file -> open" en el menu de acciones. Esto te deberìa llevar a la tìpica ventana donde puedes buscar el archivo `settings.py` y seleccionarlo. Como alternativa, puedes abrir el archivo haciendo click derecho en la carpeta djangogirls en tu escritorio. Luego, selecciona tu editor de texto en la lista. Elegir el editor es importante puesto que puede que tengas otros programas que pueden abrir el archivo pero que no te dejaran editarlo.

Sería bueno tener el horario correcto en nuestro sitio web. Ve a lista de Wikipedia de las zonas horarias y copia tu zona horaria (TZ) (e.g. `Europa/Berlín`).

En `settings.py`, encuentra la línea que contiene `TIME_ZONE` y modifícala para elegir tu zona horaria. Por ejemplo:

mysite/settings.py

```
TIME_ZONE = 'Europe/Berlin'
```

Un código de idioma tiene dos partes: el idioma, p.ej. `en` para inglés o `de` para alemán, y el código de país, p.ej. `de` para Alemania o `ch` para Suiza. Si tu idioma nativo no es el inglés, puedes añadir lo siguiente para cambiar el idioma de los botones y notificaciones de Django. Así tendrás el botón "Cancel" traducido al idioma que pongas aquí. Django trae un montón de traducciones listas para usar.

Si quieres un idioma diferente, cambia el código de idioma cambiando la siguiente línea:

mysite/settings.py

```
LANGUAGE_CODE = 'es-es'
```

También tenemos que añadir una ruta para archivos estáticos. (Veremos todo acerca de archivos estáticos y CSS más adelante.) Ve al *final* del archivo, y justo debajo de la entrada `STATIC_URL`, añade una nueva llamada `STATIC_ROOT`:

mysite/settings.py

```
STATIC_URL = '/static/'
STATIC_ROOT = os.path.join(BASE_DIR, 'static')
```

Cuando `DEBUG` es `True` y `ALLOWED_HOST` esta vacìo, el host es validado contra `['localhost', '127,0.0.1', '[::1]']`. Una vez despleguemos nuestra aplicaciòn este no sera el mismo que nuestro nombre de host en PythonAnywhere asi que cambiaremos la siguiente opciòn:

mysite/settings.py

```
ALLOWED_HOSTS = ['127.0.0.1', '.pythonanywhere.com']
```

> **Nota**: si estas usando un Chromebook, añade esta linea al final del archivo settings.py: `MESSAGE_STORAGE = 'django.contrib.messages.storage.session.SessionStorage'`
>
> Añade también `.c9users.io` a `ALLOWED_HOSTS` si estás usando cloud9

Configurar una base de datos

Hay una gran variedad de opciones de bases de datos para almacenar los datos de tu sitio. Utilizaremos la que viene por defecto, `sqlite3`.

Esta ya está configurado en esta parte de tu archivo `mysite/settings.py`:

mysite/settings.py

```
DATABASES = {
    'default': {
        'ENGINE': 'django.db.backends.sqlite3',
        'NAME': os.path.join(BASE_DIR, 'db.sqlite3'),
    }
}
```

Para crear una base de datos para nuestro blog, ejecutemos lo siguiente en la consola: `python manage.py migrate` (necesitamos estar en el directorio de `djangogirls` que contiene el archivo `manage.py`). Si eso va bien, deberías ver algo así:

command-line

```
(myvenv) ~/djangogirls$ python manage.py migrate
Operations to perform:
  Apply all migrations: auth, admin, contenttypes, sessions
Running migrations:
  Rendering model states... DONE
  Applying contenttypes.0001_initial... OK
  Applying auth.0001_initial... OK
  Applying admin.0001_initial... OK
  Applying admin.0002_logentry_remove_auto_add... OK
  Applying contenttypes.0002_remove_content_type_name... OK
  Applying auth.0002_alter_permission_name_max_length... OK
  Applying auth.0003_alter_user_email_max_length... OK
  Applying auth.0004_alter_user_username_opts... OK
  Applying auth.0005_alter_user_last_login_null... OK
  Applying auth.0006_require_contenttypes_0002... OK
  Applying auth.0007_alter_validators_add_error_messages... OK
  Applying sessions.0001_initial... OK
```

Y, ¡terminamos! Es hora de iniciar el servidor web y ver si está funcionando nuestro sitio web!

Iniciar el servidor

Debes estar en el directorio que contiene el archivo `manage.py` (en la carpeta `djangogirls`). En la consola, podemos iniciar el servidor web ejecutando `python manage.py runserver`:

command-line

```
(myvenv) ~/djangogirls$ python manage.py runserver
```

Si estas usando un Chromebook, utiliza este comando:

Cloud 9

```
(myvenv) ~/djangogirls$ python manage.py runserver 0.0.0.0:8080
```

Si estás en Windows y te falla con un error `UnicodeDecodeError`, utiliza en su lugar este comando:

command-line

```
(myvenv) ~/djangogirls$ python manage.py runserver 0:8000
```

Ahora todo lo que tienes que hacer es comprobar que tu sitio se esté ejecutando. Abre tu navegador (Firefox, Chrome, Safari, Internet Explorer o el que utilices) y escribe la dirección:

browser

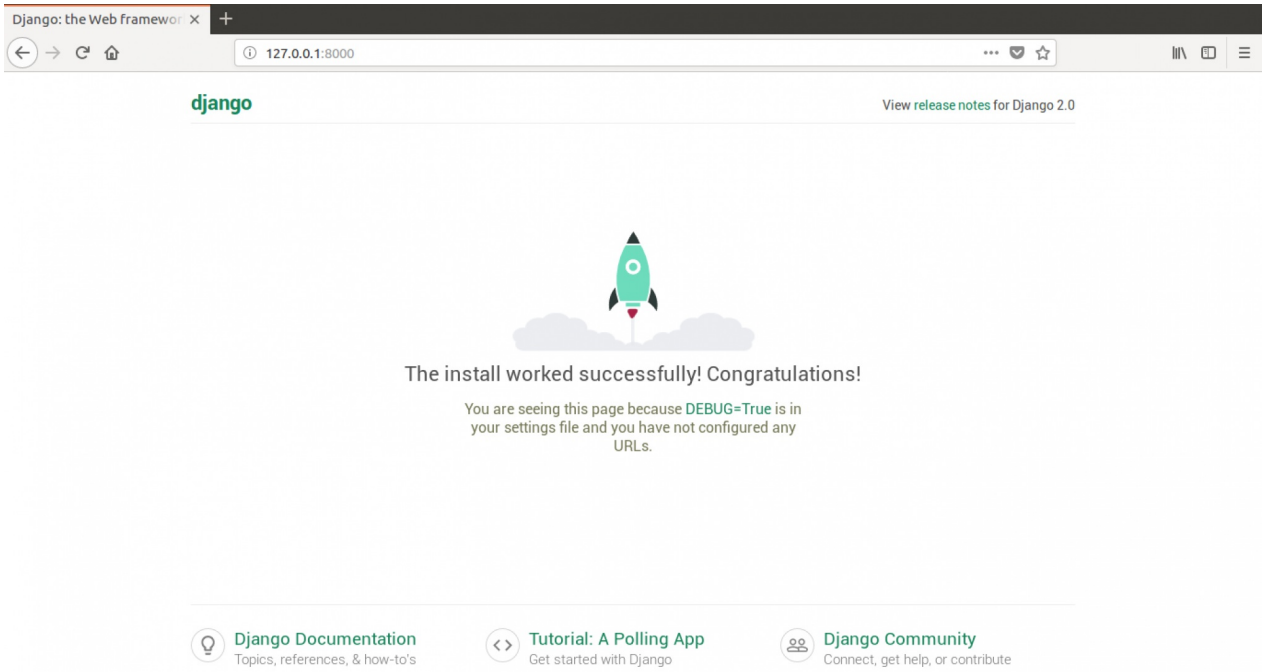
```
http://127.0.0.1:8000/
```

Si estàs usando un Chromebook, siempre visitaras tu servidor de pruebas accediendo a:

browser

```
https://django-girls-<your cloud9 username>.c9users.io
```

¡Enhorabuena! ¡Has creado tu primer sitio web y lo has iniciado usando un servidor web! ¿No es genial?



Mientras el servidor se este ejecutando, no podràs ejecutar comandos adicionales. La terminal aceptarà texto pero no ejecutara ningùn comando. Esto sucede porque el servidor se ejecuta continuamente para recibir solicitudes.

> Miramos como funcionan los servidores web en el capìtulo **Còmo funciona el internet**.

Recuerda, para escribir comandos mientras el servidor web esta funcionando, abre una nueva terminal y activa el virtualenv. Para parar el servidor web, ve a la ventana donde se esté ejecutando y pulsa CTRL+C, las teclas Control y C a la vez (en Windows puede que tengas que pulsar Ctrl+Break).

¿Preparada para el próximo paso? ¡Es momento de crear algo de contenido!

Modelos en Django

Lo que queremos crear ahora es algo que almacene todas las entradas de nuestro blog. Pero para poder hacerlo tenemos que hablar un poco sobre algo llamado `objetos`.

Objetos

Hay un concepto en el mundo de la programación llamado `programación orientada a objetos`. La idea es que en lugar de escribir todo como una aburrida secuencia de instrucciones de programación podemos modelar cosas y definir cómo interactúan entre ellas.

Entonces, ¿qué es un objeto? Es un conjunto de propiedades y acciones. Suena raro, pero te daremos un ejemplo.

Si queremos modelar un gato crearemos un objeto `Gato` que tiene algunas propiedades como: `color`, `edad`, `temperamento` (como bueno, malo, o dormilón ;)), y `dueño` (este es un objeto `Persona` o en caso de un gato callejero, esta propiedad está vacía).

Luego, el `Gato` tiene algunas acciones como: `ronronear`, `arañar` o `alimentar` (en cuyo caso daremos al gato algo de `ComidaDeGato`, el cual debería ser un objeto aparte con propiedades como `sabor`).

```
Gato
---------
color
edad
humor
dueño
ronronear()
rasguñar()
alimentarse(comida_de_gato)

ComidaDeGato
----------
sabor


ComidaDeGato
--------
sabor
```

Básicamente se trata de describir cosas reales en el código con propiedades (llamadas `propiedades del objeto`) y las acciones (llamadas `métodos`).

Y ahora, ¿cómo modelamos las entradas en el blog? Queremos construir un blog, ¿no?

Necesitamos responder a la pregunta: ¿Qué es una entrada de un blog? ¿Qué propiedades debería tener?

Bueno, seguro que nuestras entradas de blog necesitan un texto con su contenido y un título, ¿cierto? También sería bueno saber quién lo escribió, así que necesitamos un autor. Por último, queremos saber cuándo se creó y publicó la entrada.

```
Post
--------
title
text
author
created_date
published_date
```

¿Qué tipo de cosas podría hacerse con una entrada del blog? Sería bueno tener algún `método` que publique la entrada, ¿no?

Así que vamos a necesitar el método `publicar`.

Puesto que ya sabemos lo que queremos lograr, ¡podemos empezar a modelarlo en Django!

Modelos en Django

Sabiendo qué es un objeto, podemos crear un modelo en Django para nuestros entradas de blog.

Un modelo en Django es un tipo especial de objeto que se guarda en la `base de datos`. Una base de datos es una colección de datos. Es un lugar en el cual almacenarás la información sobre usuarios, tus entradas de blog, etc. Utilizaremos una base de datos SQLite para almacenar nuestros datos. Este es el adaptador de base de datos predeterminado en Django -- será suficiente para nosotros por ahora.

Puedes pensar el modelo en la base de datos, como una hoja de cálculo con columnas (campos) y filas (datos).

Crear una aplicación

Para mantener todo en orden, crearemos una aplicación separada dentro de nuestro proyecto. Es muy bueno tener todo organizado desde el principio. Para crear una aplicación, necesitamos ejecutar el siguiente comando en la consola (dentro de la carpeta de `djangogirls` donde está el archivo `manage.py`):

Mac OS X y Linux:

```
(myvenv) ~/djangogirls$ python manage.py startapp blog
```

Windows:

```
(myvenv) C:\Users\Name\djangogirls> python manage.py startapp blog
```

Notarás que se ha creado un nuevo directorio `blog` y ahora contiene una cantidad de archivos. Los directorios y archivos en nuestro proyecto deberían verse así:

```
djangogirls
├── blog
│   ├── __init__.py
│   ├── admin.py
│   ├── apps.py
│   ├── migrations
│   │   └── __init__.py
│   ├── models.py
│   ├── tests.py
│   └── views.py
├── db.sqlite3
├── manage.py
├── mysite
│   ├── __init__.py
│   ├── settings.py
│   ├── urls.py
│   └── wsgi.py
└── requirements.txt
```

Después de crear una aplicación, también necesitamos decirle a Django que debe utilizarla. Eso se hace en el fichero `mysite/settings.py` -- ábrelo en el editor. Tenemos que encontrar `INSTALLED_APPS` y agregar una línea que contiene `'blog',` justo por encima de `]`. El producto final debe tener este aspecto:

mysite/settings.py

```
INSTALLED_APPS = [
    'django.contrib.admin',
    'django.contrib.auth',
    'django.contrib.contenttypes',
    'django.contrib.sessions',
    'django.contrib.messages',
    'django.contrib.staticfiles',
    'blog',
]
```

Crear el modelo del Post

En el archivo `blog/models.py` definimos todos los objetos llamados `Models`. Este es un lugar en el cual definiremos nuestra entrada del blog.

Abre `blog/models.py` en el editor, borra todo, y escribe código como este:

blog/models.py

```
from django.db import models
from django.utils import timezone


class Post(models.Model):
    author = models.ForeignKey('auth.User', on_delete=models.CASCADE)
    title = models.CharField(max_length=200)
    text = models.TextField()
    created_date = models.DateTimeField(
            default=timezone.now)
    published_date = models.DateTimeField(
            blank=True, null=True)

    def publish(self):
        self.published_date = timezone.now()
        self.save()

    def __str__(self):
        return self.title
```

> Comprueba nuevamente que usas dos guiones bajos (`_`) en cada lado de `str`. Esta convención se usa en Python con mucha frecuencia y a veces también se llaman "dunder" (abreviatura de "double-underscore" o, en español, "doble guión bajo").

Da un poco de miedo, ¿no? Pero no te preocupes, ¡vamos a explicar qué significan estas líneas!

Todas las líneas que comienzan con `from` o `import` son líneas para agregar algo de otros archivos. Así que en vez de copiar y pegar las mismas cosas en cada archivo, podemos incluir algunas partes con `from... import ...`.

`class Post(models.Model):`, esta línea define nuestro modelo (es un `objeto`).

- `class` es una palabra clave que indica que estamos definiendo un objeto.
- `Post` es el nombre de nuestro modelo. Podemos darle un nombre diferente (pero debemos evitar espacios en blanco y caracteres especiales). Siempre inicia el nombre de una clase con una letra mayúscula.
- `models.Model` significa que Post es un modelo de Django, así Django sabe que debe guardarlo en la base de datos.

Ahora definimos las propiedades de las que hablábamos: `title`, `text`, `created_date`, `published_date` y `author`. Para ello tenemos que definir el tipo de cada campo (¿es texto? ¿un número? ¿una fecha? ¿una relación con otro objeto como un User (usuario)?)

- `models.CharField`, así es como defines un texto con un número limitado de caracteres.
- `models.TextField`, este es para texto largo sin límite. Suena perfecto para el contenido de la entrada del blog, ¿no?
- `models.DateTimeField`, este es fecha y hora.
- `modelos.ForeignKey`, este es una relación (link) con otro modelo.

No vamos a explicar aquí cada pedacito de código porque nos tomaría demasiado tiempo. Deberías echar un vistazo a la documentación de Django si quieres saber más sobre los campos de los Modelos y cómo definir cosas diferentes a las descritas anteriormente (https://docs.djangoproject.com/en/2.0/ref/models/fields/#field-types).

¿Y qué sobre `def publish(self):` ? Es exactamente el método `publish` que mencionábamos antes. `def` significa que es una función/método y `publish` es el nombre del método. Puedes cambiar el nombre del método, si quieres. La regla de nomenclatura es utilizar minúsculas y guiones bajos en lugar de espacios. Por ejemplo, un método que calcule el precio medio se podría llamar `calcular_precio_medio` .

Los métodos suelen devolver (`return` , en inglés) algo. Hay un ejemplo de esto en el método `__str__` . En este escenario, cuando llamemos a `__str__()` obtendremos un texto (**string**) con un título de Post.

También, nota que ambos `def publish(self):` , y `def __str__(self):` son indentados dentro de nuestra clase. Porque Python es sensible a los espacios en blancos, necesitamos indentar nuestros métodos dentro de la clase. De lo contrario, los métodos no pertenecen a la clase, y puedes obtener un comportamiento inesperado.

Si algo todavía no está claro sobre modelos, ¡no dudes en preguntar a tu guía! Sabemos que es complicado, sobre todo cuando aprendes lo que son funciones y objetos al mismo tiempo. Pero con suerte, ¡todo tiene un poco más de sentido para ti ahora!

Crear tablas para los modelos en tu base de datos

El último paso aquí es agregar nuestro nuevo modelo a la base de datos. Primero tenemos que hacer saber a Django que hemos hecho cambios en nuestro modelo. (Lo acabamos de crear!) Ve a tu terminal y escribe `python manage.py makemigrations blog` . Se verá así:

command-line

```
(myvenv) ~/djangogirls$ python manage.py makemigrations blog
Migrations for 'blog':
  blog/migrations/0001_initial.py:

  - Create model Post
```

Nota: Recuerda guardar los archivos que edites. De otro modo, tu computador ejecutara las versiones anteriores lo que puede ocasionar errores inesperados.

Django preparó un archivo de migración que ahora tenemos que aplicar a nuestra base de datos. Escribe `python manage.py migrate blog` y el resultado debería ser:

command-line

```
(myvenv) ~/djangogirls$ python manage.py migrate blog
Operations to perform:
  Apply all migrations: blog
Running migrations:
  Applying blog.0001_initial... OK
```

¡Hurra! ¡Nuestro modelo Post ya está en nuestra base de datos! Estaría bien verlo, ¿no? ¡Salta al siguiente capítulo para ver qué aspecto tiene tu Post!

Administrador de Django

Para agragar, editar y borrar los posts que hemos modelado, usaremos el administrador (admin) de Django.

Abre el fichero `blog/admin.py` en el editor y reemplaza su contenido con esto:

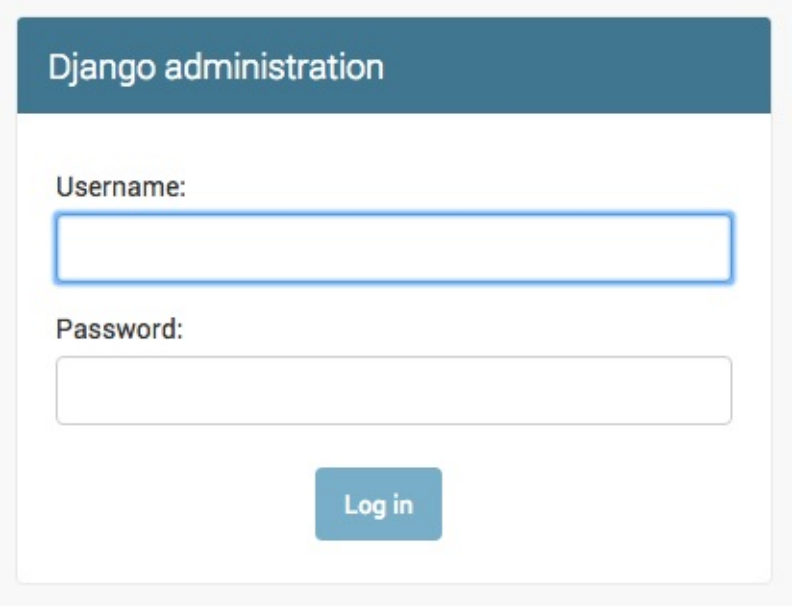
blog/admin.py

```
from django.contrib import admin
from .models import Post

admin.site.register(Post)
```

Como puedes ver, importamos (incluimos) el modelo Post definido en el capítulo anterior. Para hacer nuestro modelo visible en la página del administrador, tenemos que registrar el modelo con `admin.site.register(Post)`.

Ok, es hora de ver nuestro modelo Post. Recuerda ejecutar `python manage.py runserver` en la consola para correr el servidor web. Ve a tu navegador y escribe la dirección http://127.0.0.1:8000/admin/. Veras una página de inicio de sesión como esta:



Para iniciar sesión, deberás crear un *superusuario (superuser)*, que es un usuario que tiene control sobre todo el sitio. Vuelve a la línea de comandos, escribe `python manage.py createsuperuser` y pulsa enter.

> Recuerda, para escribir comandos mientras el servidor web esta funcionando, abre una nueva terminal y activa el virtualenv. Revisamos como escribir nuevos comandos en el capitulo **Tu primer proyecto de Django!**, al inicio de la sección **Iniciando el servidor web**.

Mac OS X o Linux:

```
(myvenv) ~/djangogirls$ python manage.py createsuperuser
```

Windows:

```
(myvenv) C:\Users\Name\djangogirls> python manage.py createsuperuser
```

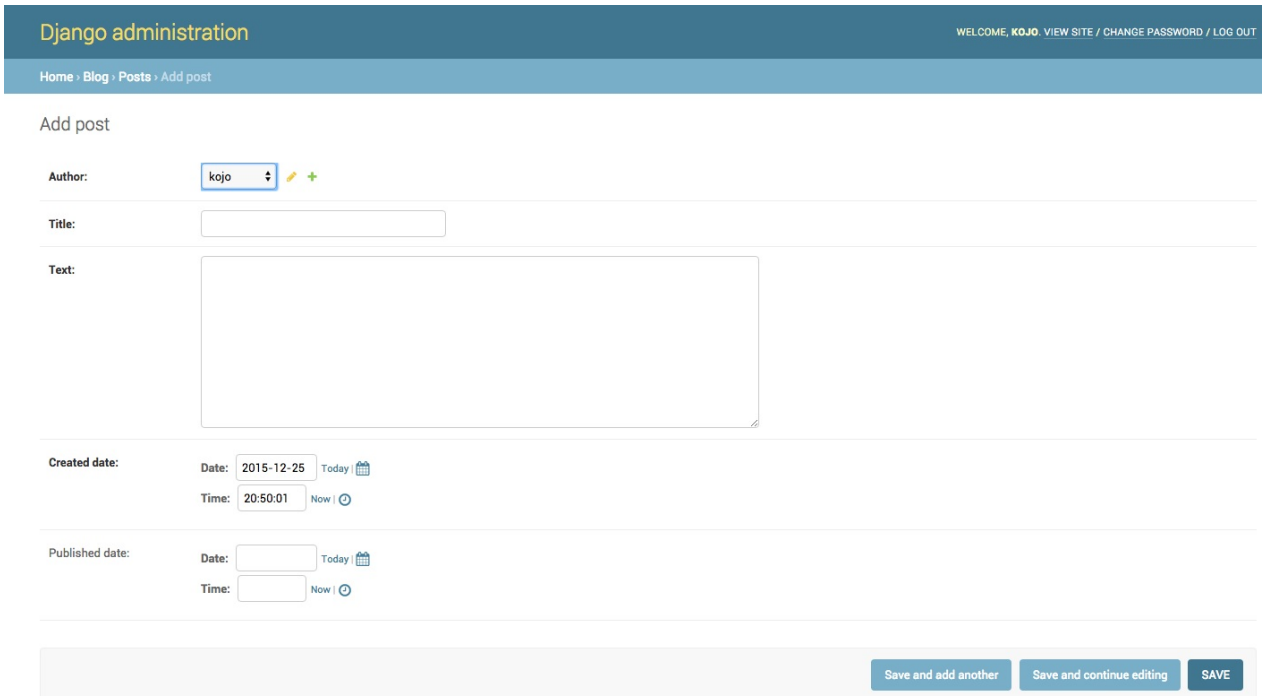
Cuando te lo pida, escribe tu nombre de usuario (en minúscula, sin espacios), email y contraseña. **No te preocupes si no puedes ver la password que estás tecleando - así es como debe ser.** Tecléalo y pulsa `intro` para continuar. Luego, verás algo así (Donde username y email serán los que escribiste anteriormente):

```
Username: admin
Email address: admin@admin.com
Password:
Password (again):
Superuser created successfully.
```

Vuelve a tu navegador. Entra con las credenciales de super usuario que tu escogiste; verás el panel de administrador de Django.

Ve a 'Posts' y curiosea un poco. Añade cinco o seis publicaciones en tu blog. No te preocupes por el contenido -- solo será visible para tí en tu ordenador -- puedes copiar y pegar texto de este tutorial para ir más rápido. :)

Asegúrate de que al menos dos o tres posts (pero no todos) tengan la fecha de publicación definida. Esto será muy poderoso después.



Si desea saber más acerca de Django admin, debería revisar la documentación Django: https://docs.djangoproject.com/en/2.0/ref/contrib/admin/

Este posiblemente sea un buen momento para tomar un café (o te) o algo para comer y re energizar tu cuerpo. Has creado tu primer modelo en Django - ¡Mereces un pequeño descanso!

¡Despliega!

> **Nota** El siguiente capítulo puede ser, a veces, un poco difícil de seguir. Ten paciencia y acábalo. El despliegue es una parte importante del proceso en el desarrollo de un sitio web. Este capítulo está a mitad del tutorial para que tu mentor pueda ayudarte a conseguir que tu sitio web esté online, algo que puede ser un poco complicado. Así, aunque se te acabe el tiempo, podrás terminar el tutorial por tu cuenta.

Hasta ahora, tu sitio web sólo está disponible en tu ordenador. ¡Ahora aprenderás como desplegarlo! El despliegue es el proceso de publicar tu aplicación en internet para que la gente pueda acceder y ver tu sitio web. :)

Como ya has aprendido, un sitio web tiene que estar en un servidor. Hay muchos proveedores de servidores disponibles en internet, nosotros vamos a usar PythonAnywhere. PythonAnywhere es gratuito para aplicaciones pequeñas que no tienen muchos visitantes, y con eso tendrás más que suficiente por ahora.

El otro servicio externo que vamos a utilizar es GitHub, un servicio de almacenamiento de código. Hay otras opciones por ahí, pero hoy en día casi todas las programadoras y programadores tenemos una cuenta de GitHub, ¡y ahora tú también la vas a tener!

Estos tres lugares serán importantes para ti. Tu ordenador local será el lugar donde desarrollas y pruebas. Cuando estés contento con los cambios, subirás una versión de tu programa a GitHub. Tu sitio web estará en PythonAnywhere y para actualizarlo descargarás la última versión de tu código desde GitHub.

Git

> **Nota** Si ya has realizado los pasos de instalación, no los tienes que repetir, puedes avanzar a la siguiente sección y empezar a crear tu repositorio de Git.

Git es un "sistema de control de versiones" que utilizan muchos programadores. Este software puede seguir los cambios realizados en archivos a lo largo del tiempo de forma que más tarde puedas volver a cualquier versión anterior. Es un poco parecido a la opción de "control de cambios" de Microsoft Word, pero mucho más potente.

Instalar Git

Installing Git: Windows
Puedes descargar Git desde git-scm.com. Puedes hacer click en "next" en todos los pasos a excepción de uno; en el quinto paso titulado "Adjusting your PATH environment", escoge "Use Git and optional Unix tools from the Windows Command Prompt" (la de abajo del todo). Aparte de eso, los valores por defecto están bien. "Checkout Windows-style, commit Unix-style line endings" también está bien.

No olvides reiniciar la terminal o powershell después de que la instalación termine.

Installing Git: OS X
Descarga Git de git-scm.com y sigue las instrucciones.

> **Nota** Si estas usando OS X 10.6, 10.7 o 10.8, tendrás que instalar git desde aquí: Git installer for OS X Snow Leopard

Installing Git: Debian or Ubuntu

command-line

```
$ sudo apt install git
```

Installing Git: Fedora

command-line

```
$ sudo dnf install git
```

Installing Git: openSUSE

command-line

```
$sudo zypper install git
```

Crear nuestro repositorio Git

Git sigue los cambios realizados a un grupo determinado de archivos en lo que llamamos un repositorio de código (abreviado "repo"). Vamos a crear uno para nuestro proyecto. Abre la consola y ejecuta los siguientes comandos en el directorio de `djangogirls`:

> **Nota** Comprueba en qué directorio estás ahora mismo (es decir, el directorio de trabajo actual) con el comando `pwd` (OSX/Linux) o `cd` (Windows) antes de inicializar el repositorio. Deberías estar en la carpeta `djangogirls`.

command-line

```
$ git init
Initialized empty Git repository in ~/djangogirls/.git/
$ git config --global user.name "Tu nombre"
$ git config --global user.email tu@ejemplo.com
```

Inicializar el repositorio de git es algo que sólo tenemos que hacer una vez por proyecto (y no tendrás que volver a teclear tu nombre de usuario y correo electrónico nunca más).

Git llevará un seguimiento de los cambios realizados en todos los archivos y carpetas en este directorio, pero hay algunos archivos que queremos que ignore. Esto lo hacemos creando un archivo llamado `.gitignore` en el directorio base. Abre tu editor y crea un nuevo archivo con el siguiente contenido:

.gitignore

```
*.pyc
*~
__pycache__
myvenv
db.sqlite3
/static
.DS_Store
```

Y guárdalo como `.gitignore` en la carpeta "djangogirls".

> **Nota** ¡El punto al principio del nombre del archivo es importante! Si tienes problemas para crearlo (a los Mac no les gusta que crees ficheros con un punto al principio del nombre usando el Finder por ejemplo), entonces usa la opción "Save As" o "Guardar como" de tu editor, esto funcionará seguro. Asegúrate de no añadir `.txt`, `.py`, o ninguna otra extensión al nombre de fichero -- Git solo lo reconocerá si se llama exactamente `.gitignore`, sin nada más.
>
> **Nota** Uno de los archivos especificados en tu `.gitignore` es `db.sqlite3`. Ese fichero es tu base de datos local, donde se almacenan los usuarios y publicaciones de tu blog. Vamos a seguir las buenas prácticas de programación web: vamos a usar bases de datos separadas para tu sitio local y tu sitio en producción en PythonAnywhere. La base de datos en PythonAnywhere podría ser SQLite, como en tu máquina de desarrollo, pero también podrías usar otro gestor de base de datos como MySQL o PostgreSQL que pueden soportar muchas más visitas que SQLite. En cualquier caso, al ignorar la base de datos SQLite en tu copia de GitHub, todos los posts y el super usuario que has creado hasta el momento solo estarán disponibles en local, y tendrás que crear nuevos usuarios y publicaciones en producción. Tu base de datos local es un buen campo de pruebas donde puedes probar diferentes cosas sin miedo a estropear o borrar las publicaciones reales de tu blog.

Te recomendamos utilizar el comando `git status` antes de `git add` o en cualquier momento en que no sepas muy bien lo que ha cambiado. Esto te ayudará a evitar sorpresas, como subir cambios o archivos que no queríamos subir. El comando `git status` muestra información sobre cualquier archivo no seguido ("untracked"), modificado ("modified"), preparado ("staged"), el estado de la rama y muchas cosas más. La salida debería ser parecida a esto:

command-line

```
$ git status
On branch master

Initial commit

Untracked files:
  (use "git add <file>..." to include in what will be committed)

        .gitignore
        blog/
        manage.py
        mysite/
        requirements.txt

nothing added to commit but untracked files present (use "git add" to track)
```

Y finalmente guardamos nuestros cambios. Ve a la consola y ejecuta estos comandos:

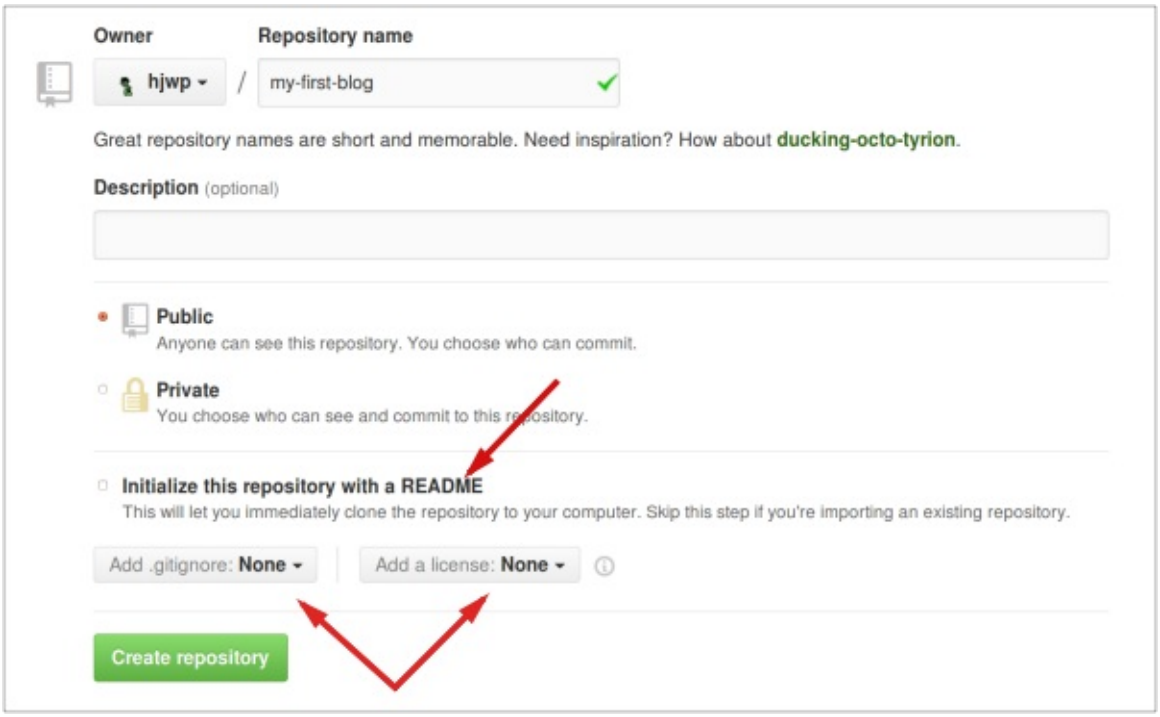
command-line

```
$ git add --all .
$ git commit -m "Mi aplicación Django Girls, primer commit"
 [...]
 13 files changed, 200 insertions(+)
 create mode 100644 .gitignore
 [...]
 create mode 100644 mysite/wsgi.py
```

Subiendo tu código a Github

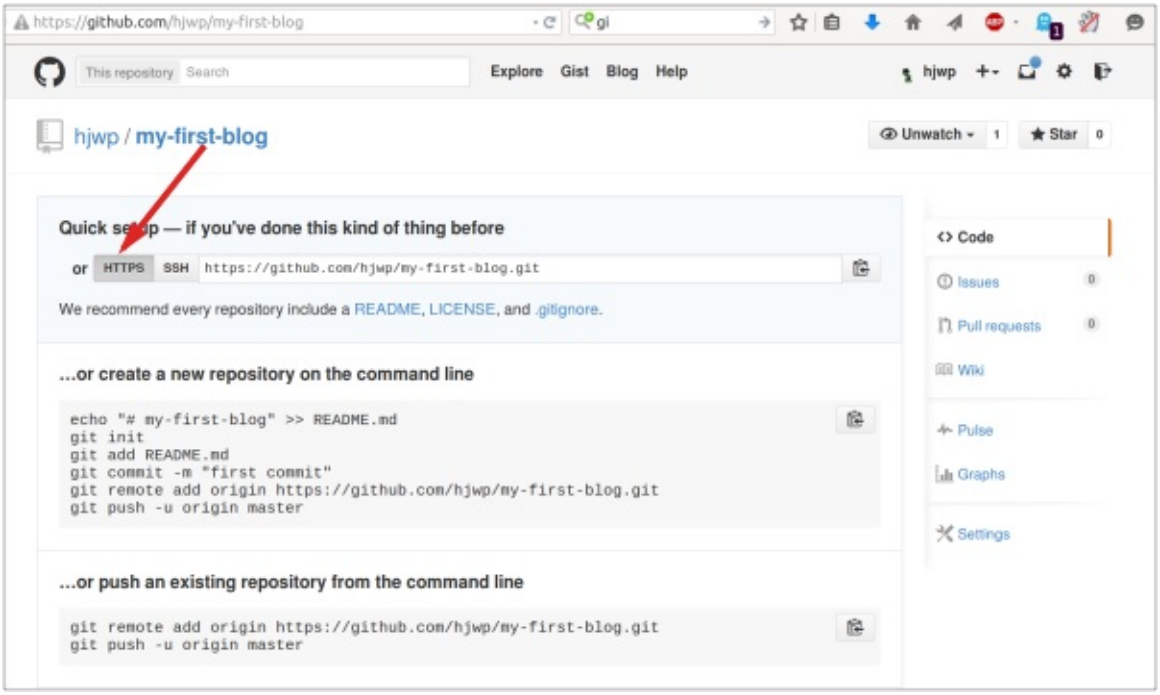
Ve a GitHub.com y regístrate para tener una cuenta gratuita. (Si ya tienes una o lo hiciste en la preparación del taller, ¡genial!)

A continuación, crea un nuevo repositorio con el nombre "my-first-blog". Deja el checkbox "initialize with a README" sin marcar, deja la opción de .gitignore vacía (ya lo hemos hecho manualmente) y deja la licencia como None.



> **Nota** El nombre `my-first-blog` es importante - podrías escoger otro, pero va a salir muchas veces en las instrucciones que vienen a continuación, y vas a tener que acordarte de cambiarlo cada vez. Lo más fácil es quedarse con el nombre `my-first-blog`.

En la siguiente pantalla, verás la URL para clonar el repo, que tendrás que usar en los comandos que van a continuación:



Ahora necesitas enlazar el repositorio Git en tu ordenador con el repositorio de GitHub.

Escribe lo siguiente en la consola (cambia `<your-github-username>` por tu nombre de usuario de GitHub, pero sin los símbolos < y > -- fíjate en que la URL debería coincidir con la URL para clonar el repo que acabas de ver):

command-line

```
$ git remote add origin https://github.com/<your-github-username>/my-first-blog.git
$ git push -u origin master
```

Cuando hagas push a GitHub, te preguntará tu usuario y password de GitHub, y después de introducirlos, deberías ver algo como esto:

command-line

```
Counting objects: 6, done.
Writing objects: 100% (6/6), 200 bytes | 0 bytes/s, done.
Total 3 (delta 0), reused 0 (delta 0)
To https://github.com/ola/my-first-blog.git

 * [new branch]      master -> master
Branch master set up to track remote branch master from origin.
```

Tu código ya está subido a GitHub. ¡Ve y compruébalo! Encontrarás que está en buena compañía - DjangoDjango Girlst Tutorial, y muchos otros proyectos de software libre están alojados en GitHub. :)

Configurar nuestro blog en PythonAnywhere

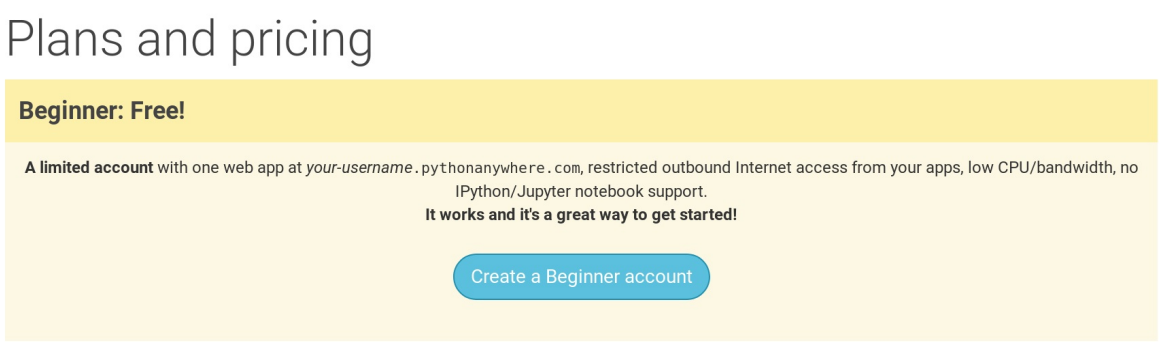
Crea una cuenta en PythonAnywhere

> **Nota** A lo mejor ya has creado una cuenta en PythonAnywhere durante los pasos de instalación. Si es así, no necesitas hacerlo otra vez.

PythonAnywhere es un servicio para ejecutar código Python en servidores "en la nube". Lo vamos a usar para alojar nuestro sitio para que esté disponible en Internet.

Crea una cuenta de Principiante ("Beginner") en PythonAnywhere (el nivel gratuito está bien, no necesitas tarjeta de crédito).

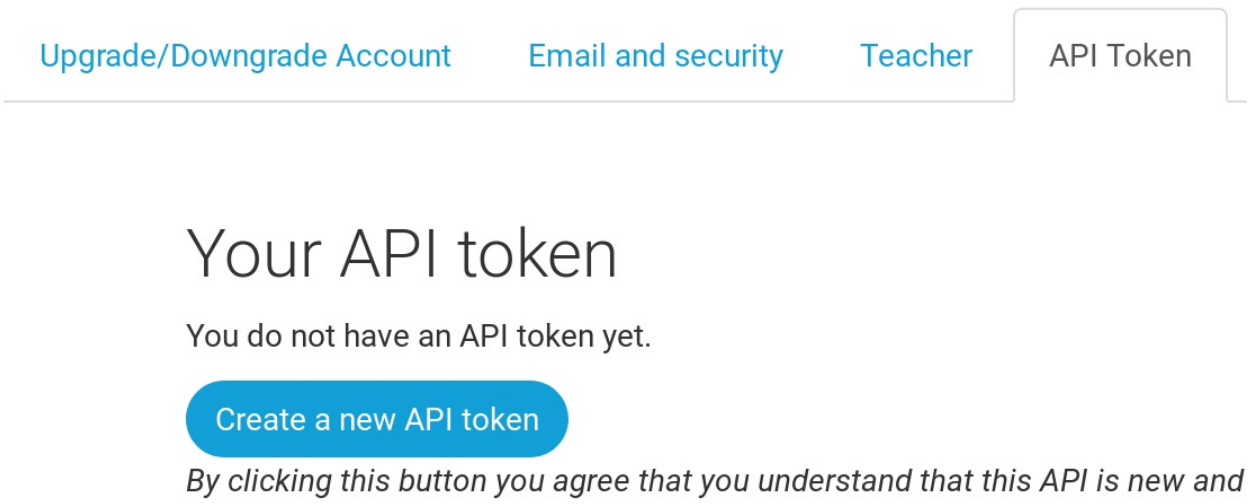
- www.pythonanywhere.com



> **Nota** Cuando elijas un nombre de usuario, recuerda que la URL de tu blog tendrá la forma `tunombredeusuario.pythonanywhere.com`, así que lo mejor será usar tu apodo o elegir un nombre que indique de qué trata tu blog.

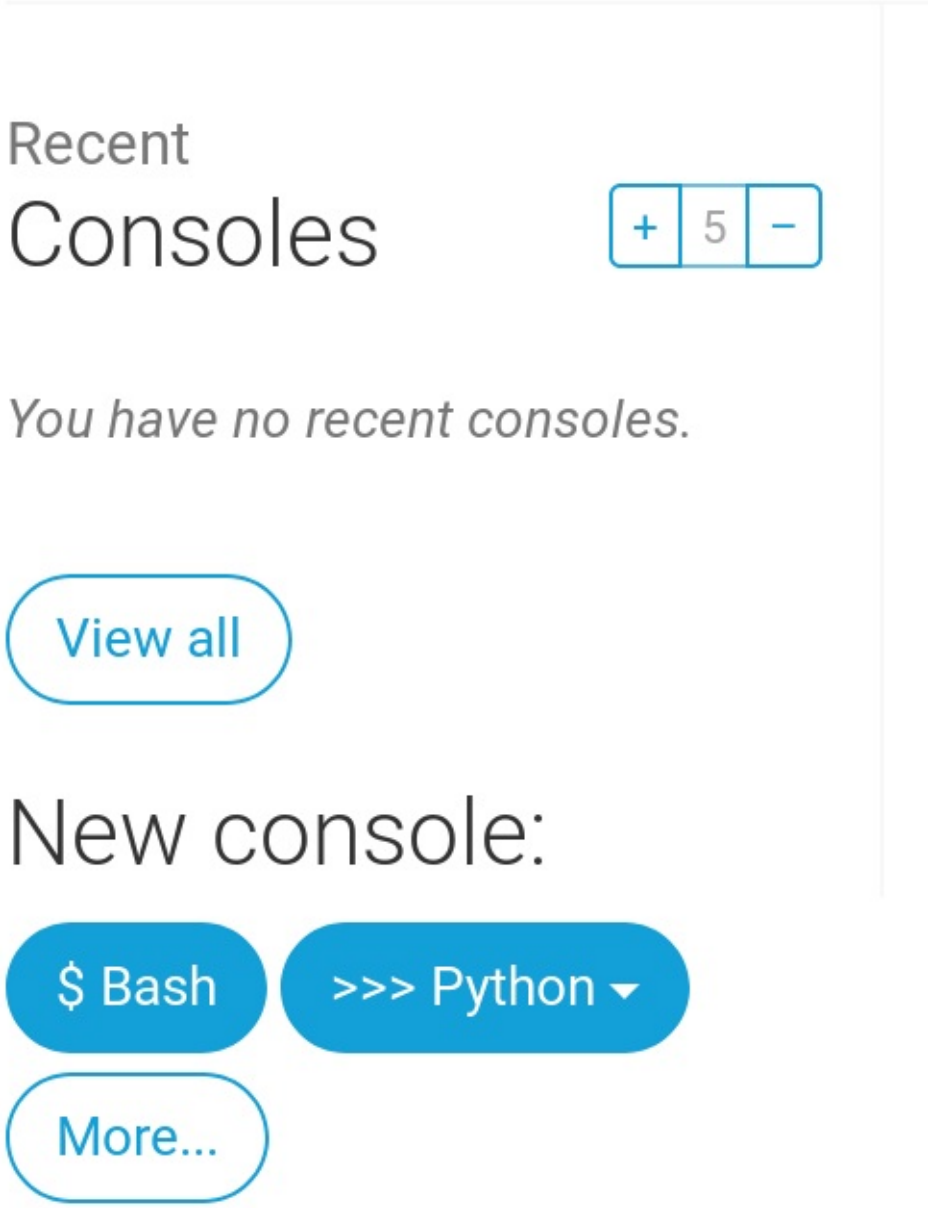
Crear un token para la API de PythonAnywhere

Solo tendrás que hacer esto la primera vez. Cuando entres en PythonAnywhere, verás el panel de control ("dashboard"). En la esquina superior derecha, busca el enlace la página de tu cuenta ("Account"), selecciona una pestaña llamada "API token" y pulsa el botón que pone "Create new API token".



Configurar nuestro sitio en PythonAnywhere

Vuelve al dashboard de PythonAnywhere haciendo click en el logo, y escoge la opción de iniciar una consola "Bash" – esta terminal es como la que hay en tu ordenador, pero en el servidor de PythonAnywhere.



> **Nota** PythonAnywhere está basado en Linux, así que si estás en Windows, la consola será un poco distinta a la de tu ordenador.

Para desplegar una aplicación web en PythonAnywhere necesitas descargar tu código de GitHub y configurar PythonAnywhere para que lo reconozca y lo sirva como una aplicación web. Hay formas de hacerlo manualmente, pero PythonAnywhere tiene una herramienta automática que lo hará todo por nosotros. Lo primero, vamos a instalar la herramienta:

PythonAnywhere command-line

```
$ pip3.6 install --user pythonanywhere
```

Eso debería mostrar en pantalla algunos mensajes como `Collecting pythonanywhere`, y finalmente una linea diciendo que ha terminado bien: `Successfully installed (...) pythonanywhere- (...)`.

Ahora ejecutaremos el asistente para configurar automáticamente nuestra aplicación desde GitHub. Teclea lo siguiente en la consola de PythonAnywhere (no te olvides de usar tu propio nombre de usuario de GitHub en lugar de `<your-github-username>`, para que la URL sea como la URL de clonar el repo de GitHub):

PythonAnywhere command-line

```
$ pa_autoconfigure_django.py https://github.com/<your-github-username>/my-first-blog.git
```

A medida que se ejecuta, podrás ver lo que hace:

- Se descarga tu código de GitHub
- Crea un virtualenv en PythonAnywhere, igual que el de tu ordenador local
- Actualiza tus ficheros de settings con algunos settings de despliegue
- Crea la base de datos en PythonAnywhere ejecutando el comando `manage.py migrate`
- Configura los archivos estáticos (static) (luego hablaremos de éstos con más detalle)
- Y configura PythonAnywhere para publicar tu aplicación web a través de su API

En PythonAnywhere todos estos pasos están automatizados, pero son los mismos que tendrías que seguir en cualquier otro proveedor de servidores.

Lo más importante que debes tener en cuenta es que tu base de datos en PythonAnywhere es totalmente independiente de la base de datos de tu propio ordenador, puede tener diferentes posts y cuentas de administrador. Como consecuencia, igual que lo hicimos en tu ordenador, tenemos que crear la cuenta de administrador con el comando `createsuperuser`. PythonAnywhere ya ha activado el virtualenv automáticamente, así que lo único que tienes que hacer es ejecutar:

PythonAnywhere command-line

```
(ola.pythonanywhere.com) $ python manage.py createsuperuser
```

Teclea las credenciales para tu usuario admin. Para evitar confusiones, te recomendamos usar el mismo nombre de usuario que usaste en tu ordenador local; aunque a lo mejor prefieres que la contraseña en PythonAnywhere sea más segura.

Ahora, si quieres, también puedes ver tu código en PythonAnywhere con el comando `ls`:

PythonAnywhere command-line

```
(ola.pythonanywhere.com) $ ls
blog  db.sqlite3  manage.py  mysite requirements.txt static
(ola.pythonanywhere.com) $ ls blog/
__init__.py  __pycache__  admin.py  forms.py  migrations  models.py  static
templates  tests.py  urls.py  views.py
```

También puedes ir a la página de ficheros ("Files") y navegar por los ficheros y directorios usando el visor de PythonAnywhere. (Desde la página de la consola ("Console"), puedes ir a cualquier otra página de PythonAnywhere usando el botón de la esquina superior derecha. Desde el resto de páginas, también hay enlaces a las otras en la parte superior.)

¡Ya estás en vivo!

¡Tu sitio ya debería estar online en internet! Haz click en la página "Web" de PythonAnywhere para obtener un enlace a él. Puedes compartir este enlace con quien tu quieras :)

> **Nota** Este es un tutorial para principiantes, y al desplegar este sitio hemos tomado algunos atajos que tal vez no sean las mejores prácticas desde el punto de vista de la seguridad. Si decides ampliar este proyecto o comenzar uno nuevo, deberías revisar el Checklist de despliegue de Django con recomendaciones para que tu sitio sea más seguro.

Consejos de depuración

Si te sale un error al ejecutar el script `pa_autconfigure_django.py`, aquí hay algunas causas comunes:

- Te has olvidado de crear el token de API de PythonAnywhere.
- No has puesto bien la URL de GitHub
- Si ves un error diciendo *"Could not find your settings.py*, es probable que no añadieras todos tus archivos a Git, y/o no los subiste a GitHub correctamente. Repasa la sección de Git más arriba

Si ves un error al visitar tu sitio, el primer lugar para ver qué está pasando es el **log de errores**. Encontrará un enlace en la página "Web" de PythonAnywhere. Mira si hay algún mensaje de error allí; los más recientes están en la parte inferior.

Hay también algunos consejos generales de depuración en la página de ayuda de PythonAnywhere.

Y recuerda, ¡tu mentor está aquí para ayudar!

¡Comprueba tu página!

La página por defecto debería decir "It worked!", tal como dice en tu ordenador local. Prueba a añadir `/admin/` al final de la URL y llegarás a la página de administración. Entra con tu usuario y password, y verás que puedes añadir nuevas publicaciones en el servidor -- recuerda, los post que tenías en tu base de datos local no se han subido a al blog publicado en producción.

Después de crear algunas publicaciones, puedes volver a tu instalación local (no la de PythonAnywhere). A partir de ahora, trabaja en tu instalación local para hacer los siguientes cambios. Este es un flujo de trabajo típico en el desarrollo web – haz cambios localmente, sube (push) esos cambios a GitHub, y descarga (pull) tus cambios al servidor de publicación. Esto te permite trabajar y experimentar en local sin romper tu página publicada. Mola, ¿eh?

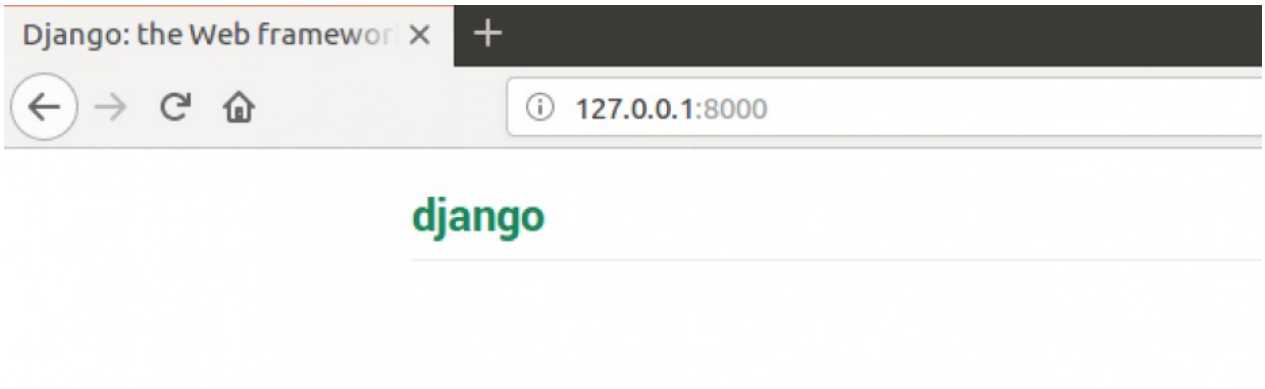
¡Date una *GRAN* palmada en la espalda! Desplegar a un servidor es una de las partes más complicadas de desarrollo web y a menudo la gente necesita varios días para que funcione del todo. Pero tu ya tienes tu sitio publicado, ¡en internet de verdad!

URLs en Django

Estamos a punto de construir nuestra primera página web: ¡una página de inicio para el blog! Pero primero, vamos a aprender un poco acerca de las urls en Django.

¿Qué es una URL?

Una URL es una dirección de la web. Puedes ver una URL cada vez que visitas una página. Se ve en la barra de direcciones del navegador. (Sì! ¡ `127.0.0.1:8000` es una URL! Y `https://djangogirls.org` también es una URL.)



Cada página en Internet necesita su propia URL. De esta manera tu aplicación sabe lo que debe mostrar a un usuario que abre una URL. En Django utilizamos algo que se llama `URLconf` (configuración de URL). URLconf es un conjunto de patrones que Django intentará comparar con la URL recibida para encontrar la vista correcta.

¿Cómo funcionan las URLs en Django?

Vamos a abrir el archivo `mysite/urls.py` en el editor de código de tu elección y veamos lo que tiene:

mysite/urls.py

```
"""mysite URL Configuration

[...]
"""
from django.urls import path, include
from django.contrib import admin

urlpatterns = [
    path('admin/', admin.site.urls),
]
```

Como puedes ver, Django ya puso algo aquí por nosotros.

Líneas entre triples comillas (`'''` o `"""`) son llamadas docstrings - puedes escribirlos en la parte superior de un archivo, clase o método para describir lo que hace. No serán ejecutadas por Python.

La URL de admin, que hemos visitado en el capítulo anterior ya está aquí:

mysite/urls.py

```
    path('admin/', admin.site.urls),
```

Esta linea dice que para cada URL que empieza con `admin/` Django encontrará su correspondiente *view*. En este caso estamos incluyendo muchas URLs admin así que no todo está empaquetado en este pequeño archivo. Es más limpio y legible.

¡Tu primera URL de Django!

¡Es hora de crear nuestra primera URL! Queremos que 'http://127.0.0.1:8000/' sea la página de inicio del blog y que muestre una lista de post.

También queremos mantener limpio el archivo `mysite/urls.py`, así que vamos a importar las urls de nuestra aplicación `blog` en el archivo principal `mysite/urls.py`.

Vamos, añade la línea para importar `blog.urls`. También tienes que cambiar la primera línea, porque vamos a usar la función `include` y tenemos que importarla.

El archivo `mysite/urls.py` debería verse ahora así:

mysite/urls.py

```
from django.urls import path, include
from django.contrib import admin

urlpatterns = [
    path('admin/', admin.site.urls),
    path('', include('blog.urls')),
]
```

Ahora Django redirigirá todo lo que entre a 'http://127.0.0.1:8000/' hacia `blog.urls` y buscará más instrucciones allí.

blog.urls

Crea un nuevo fichero vacío llamado `urls.py` el el directorio `blog`, y ábrelo en el editor de código. ¡Vale! Añade las dos primeras líneas:

blog/urls.py

```
from django.urls import path
from . import views
```

Aquí estamos importando la función de Django `path` y todos nuestras `views` desde la aplicación `blog` (no tenemos una aun, pero veremos eso en un minuto!)

Luego de esto, podemos agregar nuestro primer patrón URL:

blog/urls.py

```
urlpatterns = [
    path('', views.post_list, name='post_list'),
]
```

Como puedes ver, estamos asociando una vista (`view`) llamada `post_list` a la URL raíz. Este patrón de URL detectará la cadena vacía, y el URL resolver de Django no tiene en cuenta el nombre el dominio (i.e., http://127.0.0.1:8000/) que viene antes del path de la url. Este patrón le dirá a Django que `views.post_list` es el lugar correcto al que ir si alguien entra a tu sitio web con la dirección 'http://127.0.0.1:8000/'.

La última parte `name='post_list'` es el nombre de la URL que se utilizará para identificar a la vista. Puede coincidir con el nombre de la vista pero también puede ser algo completamente distinto. Utilizaremos las URL con nombre más delante en el proyecto así que es importante darle un nombre a cada URL de la aplicación. También deberíamos intentar mantener los

nombres de las URL únicos y fáciles de recordar.

Si tratas de visitar http://127.0.0.1:8000/ ahora, encontrarás un mensaje de error 'web page not available' a algo así. Esto es porque el servidor (¿recuerdas que escribimos `runserver` ?) ya no está funcionando. Mira la ventana de la consola del servidor para saber por qué.

```
    return _bootstrap._gcd_import(name[level:], package, level)
  File "<frozen importlib._bootstrap>", line 2254, in _gcd_import
  File "<frozen importlib._bootstrap>", line 2237, in _find_and_load
  File "<frozen importlib._bootstrap>", line 2226, in _find_and_load_unlocked
  File "<frozen importlib._bootstrap>", line 1200, in _load_unlocked
  File "<frozen importlib._bootstrap>", line 1129, in _exec
  File "<frozen importlib._bootstrap>", line 1471, in exec_module
  File "<frozen importlib._bootstrap>", line 321, in _call_with_frames_removed
  File "/Users/dana/Dana-Files/Codes/djangogirls/blog/urls.py", line 5, in <module>
    url(r'^$', views.post_list, name='post_list'),
AttributeError: 'module' object has no attribute 'post_list'
```

La consola esta mostrando un error, pero no te preocupes - de hecho es muy ùtil: està diciendote que **no existe el atributo 'post_list'**. Ese es el nombre del *view* que Django está tratando de encontrar y usar, pero aun no lo hemos creado. En esta etapa tu `/admin/` tampoco funcionarà. No te preocupes, ya llegaremos a eso. Si ves un error diferente, intenta reiniciar el servidor web. Para ello, en la consola en la que se está ejecutando el servidor web, páralo pulsando Ctrl+C (las teclas Control y C a la vez) y reinícialo ejecutando el comando `python manage.py runserver` .

> Si quieres saber más sobre Django URLconfs, mira la documentación oficial: https://docs.djangoproject.com/en/2.0/topics/http/urls/

Vistas en Django - ¡Hora de crear!

Es hora de deshacerse del error que hemos creado en el capítulo anterior! :)

Una *View* es un lugar donde ponemos la "lógica" de nuestra aplicación. Pedirá información del `modelo` que has creado antes y se la pasará a la `plantilla`. Crearemos una plantilla en el próximo capítulo. Las vistas son sólo métodos de Python que son un poco más complicados que los que escribimos en el capítulo **Introducción a Python**.

Las Vistas se colocan en el archivo `views.py`. Agregaremos nuestras *views* al archivo `blog/views.py`.

blog/views.py

Vale, abre éste fichero en el editor y mira lo que hay en él:

blog/views.py

```
from django.shortcuts import render

# Create your views here.
```

No hay demasiadas cosas aquí todavía.

Recuerda que las lineas que comienzan con `#` son comentarios - significa que Python no las ejecutarà.

Creemos una *vista (view)* como sugiere el comentario. Añade la siguiente mini-vista por debajo:

blog/views.py

```
def post_list(request):
    return render(request, 'blog/post_list.html', {})
```

Como puedes ver, hemos creado una función (`def`) llamada `post_list` que acepta `request` y `return` una función `render` que reproduce (construye) nuestra plantilla `blog/post_list.html`.

Guarda el archivo, ve a http://127.0.0.1:8000/ y mira lo que hemos hecho.

¡Otro error! Leamos lo que está pasando ahora:

Esto demuestra que el servidor está funcionando otra vez, al menos, pero todavía no se ve bien, ¿No? No te preocupes, es sólo una página de error, ¡nada a que temer! Al igual que los mensajes de error en la consola, estos son realmente muy útiles. Puedes leer que la *TemplateDoesNotExist*. Vamos a corregir este error y crear una plantilla en el próximo capítulo!

> Puedes aprender más de las vistas de Django leyendo la documentación oficial:
> https://docs.djangoproject.com/en/2.0/topics/http/views/

Introducción a HTML

Te estarás preguntando, ¿qué es una plantilla?

Una plantilla es un archivo que podemos reutilizar para presentar información diferente de forma consistente - por ejemplo, podrías utilizar una plantilla para ayudarte a escribir una carta, porque aunque cada carta puede contener un mensaje distinto y dirigirse a una persona diferente, compartirán el mismo formato.

El formato de una plantilla de Django se describe en un lenguaje llamado HTML (el HTML que mencionamos en el primer capítulo **Cómo funciona Internet**).

¿Qué es HTML?

HTML es un código simple que es interpretado por tu navegador web - como Chrome, Firefox o Safari - para mostrar una página web al usuario.

HTML significa HyperText Markup Language - en español, Lenguaje de Marcas de HyperTexto. **HyperText** -hipertexto en español- significa que es un tipo de texto que soporta hipervínculos entre páginas. **Markup** significa que hemos tomado un documento y lo hemos marcado con código para decirle a algo (en este caso, un navegador) cómo interpretar la página. El código HTML está construido con **etiquetas**, cada una comenzando con `<` y terminando con `>`. Estas etiquetas representan **elementos** de marcado.

¡Tu primera plantilla!

Crear una plantilla significa crear un archivo de plantilla. Todo es un archivo, ¿verdad? Probablemente hayas notado esto ya.

Las plantillas se guardan en el directorio de `blog/templates/blog`. Así que primero crea un directorio llamado `templates` dentro de tu directorio blog. Luego crea otro directorio llamado `blog` dentro de tu directorio de templates:

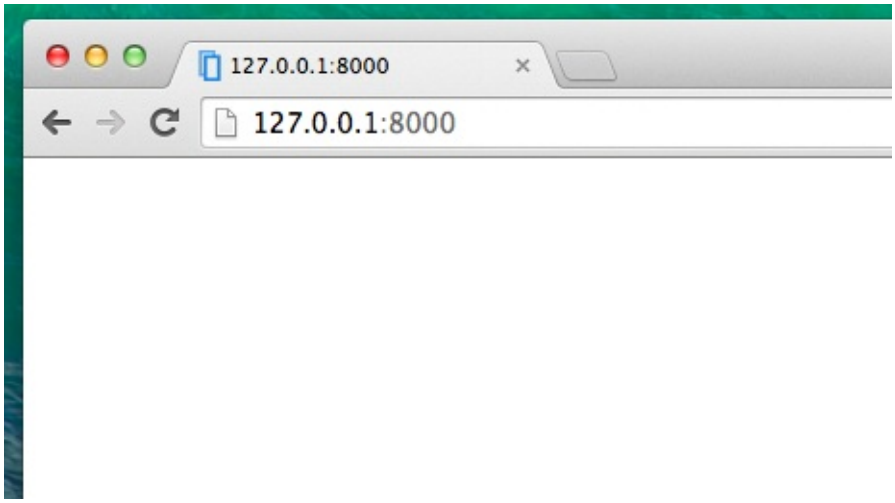
```
blog
└──templates
    └──blog
```

(Tal vez te estés preguntando por qué necesitamos dos directorios llamados `blog` – como verás más adelante, es una convención de nombres que nos facilitará la vida cuando las cosas se pongan más complicadas.)

Y ahora crea un archivo `post_list.html` (déjalo en blanco por ahora) dentro de la carpeta `blog/templates/blog`.

Mira cómo se ve su sitio web ahora: http://127.0.0.1:8000/

> Si todavía tienes un error `TemplateDoesNotExist`, intenta reiniciar tu servidor. Ve a la consola, para el servidor pulsando Ctrl+C (las teclas Control y C a la vez) y reinicialo ejecuntado el comando `python manage.py runserver`.



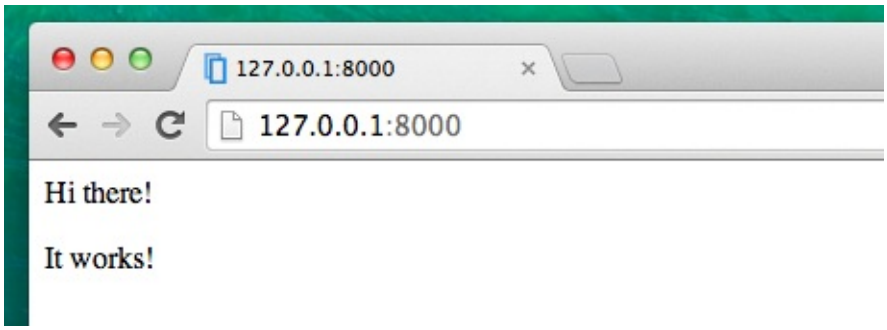
¡Ningún error más! Felicidades :) Sin embargo, por ahora, tu sitio web no está publicando nada excepto una página en blanco, porque la plantilla también está vacía. Tenemos que arreglarlo.

Abre un fichero nuevo en el editor y escribe lo siguiente:

blog/templates/blog/post_list.html

```
<html>
<body>
    <p>Hi there!</p>
    <p>It works!</p>
</body>
</html>
```

Ahora, cómo luce tu sitio web? Haz click para ver: http://127.0.0.1:8000/



¡Funcionó! Buen trabajo :)

- La etiqueta más básica, `<html>`, siempre va al inicio de cualquier página web y `</html>` va siempre al final. Como puedes ver, todo el contenido de la página web va desde el principio de la etiqueta `<html >` y hasta la etiqueta de cierre `</html >`
- `<p>` es una etiqueta para los elementos de párrafo; `</p>` cierra cada párrafo

Cabeza y cuerpo

Cada página HTML también se divide en dos elementos: **head** y **body**.

- **head** es un elemento que contiene información sobre el documento que no se muestra en la pantalla.
- **body** es un elemento que contiene todo lo que se muestra como parte de la página web.

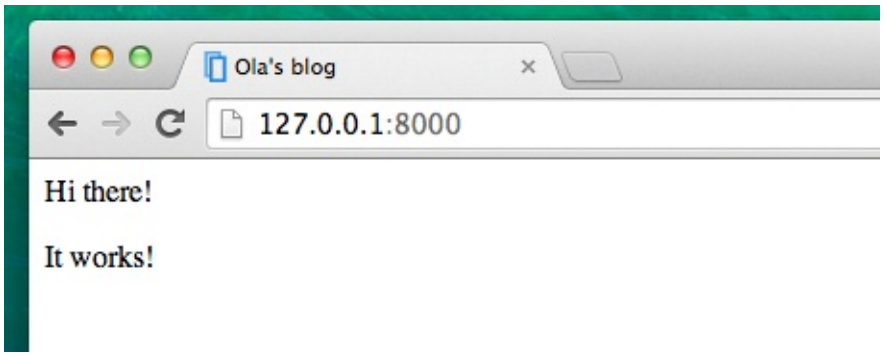
Usamos `<head>` para decirle el navegador acerca de la configuración de la página y `<body>` para decir lo que realmente está en la página.

Por ejemplo, puedes ponerle un título a la página web dentro de la `<head>`, así:

blog/templates/blog/post_list.html

```
<html>
    <head>
        <title>Ola's blog</title>
    </head>
    <body>
        <p>Hi there!</p>
        <p>It works!</p>
    </body>
</html>
```

Guarda el archivo y actualiza tu página.



¿Observas cómo el navegador ha comprendido que "Ola's blog" es el título de tu página? Ha interpretado `<title>Ola's blog</title>` y colocó el texto en la barra de título de tu navegador (también se utilizará para marcadores y así sucesivamente).

Probablemente también hayas notado que cada etiqueta de apertura coincide con una *etiqueta de cierre*, con un `/`, y que los elementos son *anidados* (es decir, no puedes cerrar una etiqueta particular hasta que todos los que estaban en su interior se hayan cerrado también).

Es como poner cosas en cajas. Tienes una caja grande, `<html></html>`; en su interior hay `<body></body>`, y que contiene las cajas aún más pequeñas: `<p></p>`.

Tienes que seguir estas reglas de etiquetas de *cierre* y de *anidación* de elementos - si no lo haces, el navegador puede no ser capaz de interpretarlos apropiadamente y tu página se mostrará incorrectamente.

Personaliza tu plantilla

¡Ahora puedes divertirte un poco y tratar de personalizar tu plantilla! Aquí hay algunas etiquetas útiles para eso:

- `<h1>Un título</h1>` - para tu título más importante
- `<h2>Un subtítulo</h2>` - para el título del siguiente nivel
- `<h3>Un subsubtítulo</h3>` - ... y así hasta `<h6>`
- `<p>Un párrafo de texto</p>`
- `<em>texto</em>` - pone en cursiva tu texto
- `<strong>texto</strong>` - pone en negrita tu texto
- `<br>` va en otra línea (no puedes poner nada dentro de br y no hay etiqueta de cierre)
- `<a href="https://djangogirls.org">link</a>` - crea un vínculo
- `<ul><li>primer elemento</li><li>segundo elemento</li></ul>` - crea una lista, ¡igual que esta!
- `<div></div>` - define una sección de la página

Aca va un ejemplo de una plantilla completa, copialo y pegalo en `blog/templates/blog/post_list.html`:

blog/templates/blog/post_list.html

```
<html>
    <head>
        <title>Django Girls blog</title>
    </head>
    <body>
        <div>
            <h1><a href="/">Django Girls Blog</a></h1>
        </div>

        <div>
            <p>published: 14.06.2014, 12:14</p>
            <h2><a href="">My first post</a></h2>
            <p>Aenean eu leo quam. Pellentesque ornare sem lacinia quam venenatis vestibulum. Donec id elit non mi po
rta gravida at eget metus. Fusce dapibus, tellus ac cursus commodo, tortor mauris condimentum nibh, ut fermentum mass
a justo sit amet risus.</p>
        </div>

        <div>
            <p>published: 14.06.2014, 12:14</p>
            <h2><a href="">My second post</a></h2>
            <p>Aenean eu leo quam. Pellentesque ornare sem lacinia quam venenatis vestibulum. Donec id elit non mi po
rta gravida at eget metus. Fusce dapibus, tellus ac cursus commodo, tortor mauris condimentum nibh, ut f.</p>
        </div>
    </body>
</html>
```

Aquí hemos creado tres secciones `div`.

- El primer elemento `div` contiene el título de nuestro blog - es un encabezado y un enlace
- Otros dos elementos `div` contienen nuestras publicaciones (posts) del blog con su fecha de publicación, `h2` con el título del post que es clickable y dos `p`s (párrafos) de texto, uno para la fecha y otro para el contenido del post.

Nos da este efecto:

¡Yaaay! Pero hasta el momento, nuestra plantilla sólo muestra exactamente **la misma información** - considerando que antes hablábamos de plantillas como permitiéndonos mostrar información **diferente** en el **mismo formato**.

Lo que queremos realmente es mostrar posts reales añadidos en nuestra página de administración de Django - y ahí es a donde vamos a continuación.

Una cosa más: ¡despliega!

Sería bueno ver todo esto disponible en Internet, ¿no? Hagamos otro despliegue en PythonAnywhere:

Haz commit, y sube tu código a GitHub

En primer lugar, vamos a ver qué archivos han cambiado desde la última puesta en marcha (ejecute estos comandos localmente, no en PythonAnywhere):

command-line

```
$ git status
```

Asegúrate de que estás en el directorio `djangogirls` y vamos a decirle a `git` que incluya todos los cambios en este directorio:

command-line

```
$ git add --all .
```

> **Nota** `--all` significa que `git` tambien reconocera si has borrado archivos (por defecto, solo reconoce archivos nuevos o modificados). También recuerda (del capítulo 3) que `.` significa el directorio actual.

Antes de que subamos todos los archivos, vamos a ver qué es lo que `git` subirá (todos los archivos que `git` cargará deberían aparecer en verde):

command-line

```
$ git status
```

Ya casi estamos, ahora es tiempo de decirle que guarde este cambio en su historial. Vamos a darle un "mensaje de commit" donde describimos lo que hemos cambiado. Puedes escribir cualquier cosa que te gustaría en esta etapa, pero es útil escribir algo descriptivo para que puedes recordar lo que has hecho en el futuro.

command-line

```
$ git commit -m "Cambie el HTML para la página."
```

> **Nota** Asegúrate de usar comillas dobles alrededor del mensaje de commit.

Una vez hecho esto, subimos (push) los cambios a Github:

command-line

```
$ git push
```

Descarga tu nuevo código a PythonAnywhere y actualiza tu aplicación web

- Abre la página de consolas de PythonAnywhere y ve a tu **consola Bash** (o comienza una nueva). Luego, ejecuta:

PythonAnywhere command-line

```
$ cd ~/<your-pythonanywhere-username>.pythonanywhere.com
$ git pull
[...]
```

(Recuerda sustituir `<your-pythonanywhere-username>` por tu nombre de usuario real en PythonAnywhere sin los < >).

Y mira como se descarga tu código. Si quieres comprobar que efectivamente ha llegado bien, puedes ir a la **página "Files"** y ver tu código en PythonAnywhere (puedes ir a otras páginas de PythonAnywhere desde el botón de la esquina superior derecha de la página de la consola).

- Finalmente, ve a la página "Web" y pulsa **Reload** en tu aplicación web.

¡Tu nueva versión ya debería estar publicada! Ve al navegador y refresca tu sitio web. Deberías ver los cambios. :)

ORM de Django y QuerySets

En este capítulo aprenderás cómo Django se conecta a la base de datos y almacena los datos en ella. ¡Vamos a sumergirnos!

¿Qué es un QuerySet?

Un QuerySet es, en esencia, una lista de objetos de un modelo determinado. Un QuerySet te permite leer los datos de la base de datos, filtrarlos y ordenarlos.

Es más fácil de aprender con ejemplos. Vamos a intentarlo, ¿de acuerdo?

Django shell

Abre tu consola local (no la de PythonAnywhere) y escribe este comando:

command-line

```
(myvenv) ~/djangogirls$ python manage.py shell
```

El resultado debería ser:

command-line

```
(InteractiveConsole)
>>>
```

Ahora estás en la consola interactiva de Django. Es como una consola de Python normal, pero con un poco de magia de Django. :) Aquí también se pueden usar todos los comandos de Python.

Todos los objetos

Vamos a mostrar todos nuestros posts primero. Puedes hacerlo con el siguiente comando:

command-line

```
>>> Post.objects.all()
Traceback (most recent call last):
      File "<console>", line 1, in <module>
NameError: name 'Post' is not defined
```

¡Uy! Apareció un error. Nos dice que Post no existe. Esto es correcto, ¡olvidamos importarlo!

command-line

```
>>> from blog.models import Post
```

Vamos a importar el modelo `Post` de `blog.models`. Y probamos de nuevo a mostrar todas las publicaciones (posts):

command-line

```
>>> Post.objects.all()
<QuerySet [<Post: my post title>, <Post: another post title>]>
```

¡Es la lista de posts que creamos anteriormente! Creamos estos posts usando la interfaz de administración de Django. Pero, ahora queremos crear nuevos posts usando Python, ¿cómo lo hacemos?

Crear objetos

Esta es la forma de crear un nuevo objeto Post en la base de datos:

command-line

```
>>> Post.objects.create(author=me, title='Sample title', text='Test')
```

Pero nos falta un ingrediente aquí: `me`. Tenemos que pasar una instancia del modelo `User` como autor. ¿Eso cómo se hace?

Primero importemos el modelo User:

command-line

```
>>> from django.contrib.auth.models import User
```

¿Qué usuarios tenemos en nuestra base de datos? Prueba esto:

command-line

```
>>> User.objects.all()
<QuerySet [<User: ola>]>
```

¡Este es el superusuario que hemos creado antes! Ahora, vamos a obtener una instancia de éste usuario (cambia el código para usar tu propio nombre de usuario):

command-line

```
>>> me = User.objects.get(username='ola')
```

Como vés, ya hemos obtenido (`get`) un usuario (`User`) cuyo `username` es igual a 'ola'. ¡Mola!

Ahora, finalmente, podemos crear nuestra entrada:

command-line

```
>>> Post.objects.create(author=me, title='Sample title', text='Test')
<Post: Sample title>
```

¡Hurra! ¿Quieres probar si funcionó?

command-line

```
>>> Post.objects.all()
<QuerySet [<Post: my post title>, <Post: another post title>, <Post: Sample title>]>
```

¡Ahí está, una entrada de blog más en la lista!

Agrega más entradas

Ahora puedes divertirte un poco y agregar más entradas para ver cómo funciona. Agrega dos o tres más y avanza a la siguiente parte.

Filtrar objetos

Una parte importante de los QuerySets es la habilidad para filtrar los resultados. Digamos que queremos encontrar todos los post del usuario ola. Usaremos `filter` en vez de `all` en `Post.objects.all()`. Entre paréntesis estableceremos qué condición (o condiciones) debe cumplir un post del blog para aparecer como resultado en nuestro queryset. En nuestro caso sería `author` es igual a `me`. La forma de escribirlo en Django es: `author=me`. Ahora nuestro bloque de código tiene este aspecto:

command-line

```
>>> Post.objects.filter(author=me)
<QuerySet [<Post: Sample title>, <Post: Post number 2>, <Post: My 3rd post!>, <Post: 4th title of post>]>
```

¿O quizá queremos ver todas las entradas que contengan la palabra 'title' en el campo `title`?

command-line

```
>>> Post.objects.filter(title__contains='title')
<QuerySet [<Post: Sample title>, <Post: 4th title of post>]>
```

> **Nota** Hay dos guiones bajos (`_`) entre `title` y `contains`. El ORM de Django utiliza esta sintaxis para separar los nombres de los campos ("title") de las operaciones o filtros ("contains"). Si sólo utilizas un guión bajo, obtendrás un error como "FieldError: Cannot resolve keyword title_contains".

También puedes obtener una lista de todos los post publicados. Lo hacemos filtrando los post que tienen la fecha de publicación, `published_date`, en el pasado:

command-line

```
>>> from django.utils import timezone
>>> Post.objects.filter(published_date__lte=timezone.now())
<QuerySet []>
```

Por desgracia, el post que hemos añadido desde la consola de Python aùn no está publicado. Pero lo podemos cambiar! Primero obtèn una instancia de la entrada que queremos publicar:

command-line

```
>>> post = Post.objects.get(title="Sample title")
```

Y luego publícala con nuestro método `publish`:

command-line

```
>>> post.publish()
```

Ahora vuelve a intentar obtener la lista de posts publicados (pulsa la tecla de "flecha arriba" tres veces y pulsa `enter`):

command-line

```
>>> Post.objects.filter(published_date__lte=timezone.now())
<QuerySet [<Post: Sample title>]>
```

Ordenar objetos

Los QuerySets también te permiten ordenar la lista de objetos. Intentemos ordenarlos por el campo `created_date`:

command-line

```
>>> Post.objects.order_by('created_date')
<QuerySet [<Post: Sample title>, <Post: Post number 2>, <Post: My 3rd post!>, <Post: 4th title of post>]>
```

También podemos invertir el orden agregando `-` al principio:

command-line

```
>>> Post.objects.order_by('-created_date')
<QuerySet [<Post: 4th title of post>,  <Post: My 3rd post!>, <Post: Post number 2>, <Post: Sample title>]>
```

Encadenar QuerySets

También puedes combinar QuerySets **encadenando** uno con otro:

```
>>> Post.objects.filter(published_date__lte=timezone.now()).order_by('published_date')
<QuerySet [<Post: Post number 2>, <Post: My 3rd post!>, <Post: 4th title of post>, <Post: Sample title>]>
```

Es muy potente y te permite escribir consultas bastante complejas.

¡Genial! ¡Ahora estás lista para la siguiente parte! Para cerrar la consola, escribe esto:

command-line

```
>>> exit()
$
```

Datos dinámicos en plantillas

Tenemos diferentes piezas en su lugar: el modelo `Post` está definido en `models.py`, tenemos a `post_list` en `views.py` y la plantilla agregada. ¿Pero cómo haremos realmente para que nuestros posts aparezcan en nuestra plantilla HTML? Porque eso es lo que queremo, tomar algún contenido (modelos guardados en la base de datos) y mostrarlo adecuadamente en nuestra plantilla, ¿no?

Esto es exactamente lo que las *views* se supone que hacen: conectar modelos con plantillas. En nuestra *view* `post_list` necesitaremos tomar los modelos que deseamos mostrar y pasarlos a una plantilla. En una *vista* decidimos que (modelo) se mostrará en una plantilla.

Muy bien, entonces ¿cómo lo logramos?

Tenemos que abrir `blog/views.py` en el editor. De momento `post_list` *view* tiene esto:

blog/views.py

```
from django.shortcuts import render

def post_list(request):
    return render(request, 'blog/post_list.html', {})
```

¿Recuerdas cuando hablamos de incluir código en diferentes archivos? Ahora tenemos que incluir el modelo que definimos en el archivo `models.py`. Agregaremos la línea `from .models import Post` de la siguiente forma:

blog/views.py

```
from django.shortcuts import render
from .models import Post
```

El punto antes de `models` indica el *directorio actual* o la *aplicación actual*. Ambos, `views.py` y `models.py` están en el mismo directorio. Esto significa que podemos utilizar `.` y el nombre del archivo (sin `.py`). Ahora importamos el nombre del modelo (`Post`).

¿Pero ahora qué sigue? Para tomar posts reales del modelo `Post`, necesitamos algo llamado `QuerySet`.

QuerySet

Ya debes estar familiarizada con la forma en que funcionan los QuerySets. Hablamos de ellos en el capítulo Django ORM (QuerySets).

Así que ahora nos interesa tener una lista de post publicados ordenados por `published_date` (fecha de publicación), ¿no? ¡Ya lo hicimos en el capítulo QuerySets!

blog/views.py

```
Post.objects.filter(published_date__lte=timezone.now()).order_by('published_date')
```

Abre `blog/views.py` en el editor, y añade este trozo de código a la función `def post_list(request)` -- pero no te olvides de añadir `from django.utils import timezone` antes:

blog/views.py

```
from django.shortcuts import render
from django.utils import timezone
from .models import Post

def post_list(request):
    posts = Post.objects.filter(published_date__lte=timezone.now()).order_by('published_date')
    return render(request, 'blog/post_list.html', {})
```

La última parte es pasar el QuerySet `posts` a la plantilla context. No te preocupes, mostraremos como mostrarlo más adelante.

Fíjate en que creamos una *variable* para el QuerySet: `posts`. Trátala como si fuera el nombre de nuestro QuerySet. De aquí en adelante vamos a referirnos al QuerySet con ese nombre.

En la función `render` tenemos el parámetro `request` (todo lo que recibimos del usuario via Internet) y otro parámetro dándole el archivo de la plantilla (`'blog/post_list.html'`). El último parámetro, que se ve así: `{}` es un lugar en el que podemos agregar algunas cosas para que la plantilla las use. Necesitamos nombrarlos (los seguiremos llamando `'posts'` por ahora). :) Se debería ver así: `{'posts': posts}`. Fíjate en que la parte antes de `:` es una cadena; tienes que envolverla con comillas: `"`.

Finalmente nuestro archivo `blog/views.py` debería verse así:

blog/views.py

```
from django.shortcuts import render
from django.utils import timezone
from .models import Post

def post_list(request):
    posts = Post.objects.filter(published_date__lte=timezone.now()).order_by('published_date')
    return render(request, 'blog/post_list.html', {'posts': posts})
```

¡Terminamos! Ahora regresemos a nuestra plantilla y mostremos este QuerySet.

Si quieres leer un poco más acerca de QuerySets en Django, puedes darle un vistazo a: https://docs.djangoproject.com/en/2.0/ref/models/querysets/

Plantillas de Django

¡Es hora de mostrar algunos datos! Para ello Django incorpora unas etiquetas de plantillas, **template tags**, muy útiles.

¿Qué son las etiquetas de plantilla?

Verás, en HTML no se puede escribir código en Python porque los navegadores no lo entienden. Sólo saben HTML. Sabemos que HTML es bastante estático, mientras que Python es mucho más dinámico.

Las **etiquetas de plantilla de Django** nos permiten insertar elementos de Python dentro del HTML, para que puedas construir sitios web dinámicos más rápida y fácilmente. ¡Genial!

Mostrar la plantilla lista de posts

En el capítulo anterior le dimos a nuestra plantilla una lista de entradas en la variable `posts`. Ahora la vamos a mostrar en HTML.

Para imprimir una variable en una plantilla de Django, utilizamos llaves dobles con el nombre de la variable dentro, algo así:

blog/templates/blog/post_list.html

```
{{ posts }}
```

Prueba esto en la plantilla `blog/templates/blog/post_list.html`. Ábrela en el editor de código, y cambia todo desde el segundo `<div>` hasta el tercer `</div>` por `{{ posts }}`. Guarda el archivo y refresca la página para ver los resultados:

Como puedes ver, lo que hemos conseguido es esto:

blog/templates/blog/post_list.html

```
<QuerySet [<Post: My second post>, <Post: My first post>] >
```

Significa que Django lo entiende como una lista de objetos. ¿Recuerdas de **Introducción a Python** cómo podemos mostrar listas? Sí, ¡con bucles for! En una plantilla de Django se hacen así:

blog/templates/blog/post_list.html

```
{% for post in posts %} {{ post }}{% endfor %}
```

Prueba esto en tu plantilla.

¡Funciona! Pero queremos que se muestren como los post estáticos que creamos anteriormente en el capítulo de **Introducción a HTML**. Usted puede mezclar HTML y etiquetas de plantilla. Nuestro `body` se verá así:

blog/templates/blog/post_list.html

```
< div >< h1 >< a href = "/" > Django chicas Blog < /a >< / h1 >< / div >{% for post in posts %} < div >< p > publicado: {{ post.published_date }} < /p >< h1 >< a href = "" >{{ post.title }} < /a >< / h1 >< p >{{ post.text|linebreaksbr }} < /p >< / div >{% endfor %}
```

Todo lo que pongas entre `{% for %}` y `{% endfor %}` se repetirá para cada objeto de la lista. Refresca la página:

¿Has notado que utilizamos una notación diferente esta vez (`{{ post.title }}` o `{{ post.text }}`)? Estamos accediendo a los datos en cada uno de los campos definidos en nuestro modelo `Post` . También el `|linebreaksbr` está pasando el texto de los post a través de un filtro para convertir saltos de línea en párrafos.

Una cosa más

Sería bueno ver si tu sitio web seguirá funcionando en la Internet pública, ¿no? Vamos a intentar desplegar de nuevo en PythonAnywhere. Aquí va un resumen de los pasos…

- Lo primero, sube tu código a GitHub

command-line

```
$ git status
[...]
$ git add --all .
$ git status
[...]
$ git commit -m "Templates modificados para mostrar post desde base de datos."
[...]
$ git push
```

- Luego, vuelve a entrar en PythonAnywhere y ve a tu **consola Bash** (o inicia una nueva), y ejecuta:

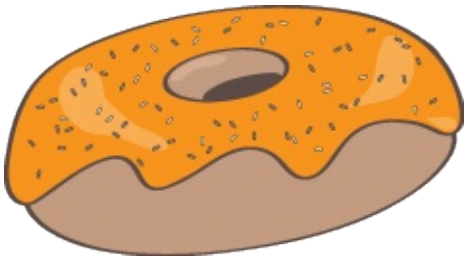
PythonAnywhere command-line

```
$ cd $USER.pythonanywhere.com
$ git pull
[...]
```

- Y finalmente, ve a la página "Web" y haz click en **Reload** en tu aplicación web. (Para ir a otras páginas de PythonAnywhere desde la consola, haz click en el botón de la esquina superior derecha.) Los cambios deberían estar visibles en https://yourname.pythonanywhere.com -- ¡compruébalo en en navegador! Si ves distintas publicaciones en el sitio en PythonAnywhere de las que tienes en tu servidor local, es lo normal. Tienes dos bases de datos, una en tu ordenador local y otra en PythonAnywhere y no tienen por qué tener el mismo contenido.

¡Felicidades! Ahora intenta añadir un nuevo post en tu administrador de Django (recuerda añadir published_date!) Asegurate de que estas en el administrador de Django de pytonanywhere, https://tunombre.pythonanywhere.com/admin. Luego actualiza tu página para ver si los post aparecen.

¿Funciona de maravilla? ¡Estamos orgullosas! Aléjate un rato del ordenador, te has ganado un descanso. :)



CSS - ¡Que quede bonito!

Nuestro blog todavía es un poco feo, ¿no te parece? ¡Es hora de ponerlo bonito! Para eso, vamos a usar CSS.

¿Qué es CSS?

El lenguaje CSS (las siglas en inglés de hojas de estilos en cascada, o Cascading Style Sheets) sirve para describir la apariencia de un sitio web escrito en un lenguaje de marcado (como HTML). Es como la capa de pintura para nuestra página web. ;)

Pero no queremos empezar de cero otra vez, ¿verdad? De nuevo vamos a usar algo que otros programadores han publicado ya en Internet. Estar siempre reinventando la rueda no mola.

¡Vamos a usar Bootstrap!

Bootstrap es uno de los frameworks de HTML y CSS mas populares para desarrollar para sitios web atractivos: https://getbootstrap.com/

Lo escribieron programadores que trabajaban en Twitter. ¡Ahora lo mantienen y desarrollan voluntarios de todo el mundo!

Instalar Bootstrap

Para instalar Bootstrap, abre tu fichero `.html` en el editor de código y añade esto a la sección `<head>` :

blog/templates/blog/post_list.html

```
<link rel="stylesheet" href="//maxcdn.bootstrapcdn.com/bootstrap/3.2.0/css/bootstrap.min.css">
<link rel="stylesheet" href="//maxcdn.bootstrapcdn.com/bootstrap/3.2.0/css/bootstrap-theme.min.css">
```

Esto no añade ningún archivo a tu proyecto. Solo apunta a archivos que existen en Internet. Así que adelante, accede a tu sitio web y refresca la página ¡Aquí la tienes!

¡Ya tiene mejor pinta!

Archivos estáticos (static files) en Django

Finalmente nos vamos a fijar en esto que hemos estado llamando **archivos estáticos**. Los archivos estáticos son los archivos CSS e imágenes. Su contenido no depende del contexto de la petición y siempre será el mismo para todos los usuarios.

Dónde poner los archivos estáticos en Django

Django ya sabe dónde encontrar los archivos estáticos de la app "admin". Ahora necesitamos añadir los archivos estáticos de nuestra aplicación, `blog`.

Crearemos una carpeta llamada `static` dentro de la app blog:

```
djangogirls
├── blog
│   ├── migrations
│   ├── static
│   └── templates
└── mysite
```

Django encontrará automáticamente cualquier carpeta llamada "static" dentro de cualquiera de las carpetas de tus apps. Podrá usar su contenido como archivos estáticos.

¡Tu primer archivo CSS!

Vamos a crear un archivo CSS, para añadir tu propio estilo a la página. Crea un nuevo directorio llamado `css` dentro de la carpeta `static`. A continuación, crea un nuevo archivo llamado `blog.css` dentro de la carpeta `css`. ¿Listos?

```
djangogirls
└── blog
     └── static
          └── css
               └── blog.css
```

¡Vamos a escribir algo de CSS! Abre el archivo `blog/static/css/blog.css` en el editor de código.

No vamos a profundizar demasiado en cómo personalizar y aprender CSS. Pero te podemos recomendar un un curso gratuito de CSS al final de esta página por si quieres aprender más.

Pero vamos a hacer al menos un poco. ¿Qué te parece si cambiamos el color del título? Los ordenadores utilizan códigos especiales para expresar los colores. Estos códigos empiezan con `#` seguidos por 6 letras (A-F) y números (0-9). Por ejemplo, el código del color azul es `#0000FF`. Puedes encontrar los códigos de muchos colores aquí: http://www.colorpicker.com/ y en otras páginas web. También puedes utilizar colores predefinidos utilizando su nombre en inglés, como `red` y `green`.

En el archivo `blog/static/css/blog.css` deberías añadir el siguiente código:

blog/static/css/blog.css

```
h1 a {
    color: #FCA205;
}
```

`h1 a` es un selector CSS. Esto significa que se va a aplicar el estilo a cualquier elemento `a` que esté dentro de un elemento `h1`. Así, cuando tenemos algo como `<h1><a href="">link</a></h1>`, se aplicará el estilo `h1 a`. En este caso le estamos diciendo que cambie el color a `#FCA205`, que es naranja. ¡O puedes poner el color que tu quieras!

En un archivo CSS se definen los estilos de los elementos que aparecen en el archivo HTML. La primera forma de identificar los elementos es por su nombre. Puede que los recuerdes como 'tags' de la sección de HTML. Cosas como `a`, `h1`, y `body` son algunos ejemplos de nombres de elementos. También podemos identificar elementos por el atributo

`class` o el atributo `id` . Los valores de "class" e "id" son nombres que das al elemento para poderlo identificar. Con el atributo "class" identificamos grupos de elementos del mismo tipo y con el atributo "id" identificamos un elemento específico. Por ejemplo, el siguiente elemento lo podrías identificar por su nombre de "tag" `a` , por su "class" `external_link` , o por su "id" `link_to_wiki_page` :

```
<a href="https://en.wikipedia.org/wiki/Django" class="external_link" id="link_to_wiki_page">
```

Si quieres aprender más sobre los selectores CSS puedes consultar en Selectores de CSS en w3schools.

También necesitamos decirle a nuestra plantilla HTML que hemos añadido código CSS. Abre el archivo `blog/templates/blog/post_list.html` en el editor de código y añade esta línea al principio del todo:

blog/templates/blog/post_list.html

```
{% load static %}
```

Aquí solo estamos cargando archivos estáticos. :) Entre las etiquetas `<head>` y `</head>` , después de los enlaces a los archivos CSS de Bootstrap, añade esta línea:

blog/templates/blog/post_list.html

```
<link rel="stylesheet" href="{% static 'css/blog.css' %}">
```

El navegador lee los archivos en el orden que le son dados, por lo que debemos asegurarnos de que está en el lugar correcto. De lo contrario, el código en nuestro archivo podría ser reemplazado por código en nuestros archivos Bootstrap. Le acabamos de decir a nuestra plantilla dónde se encuentra nuestro archivo CSS.

Ahora tu archivo debe tener este aspecto:

blog/templates/blog/post_list.html

```
{% load static %}
<html>
    <head>
        <title>Django Girls blog</title>
        <link rel="stylesheet" href="//maxcdn.bootstrapcdn.com/bootstrap/3.2.0/css/bootstrap.min.css">
        <link rel="stylesheet" href="//maxcdn.bootstrapcdn.com/bootstrap/3.2.0/css/bootstrap-theme.min.css">
        <link rel="stylesheet" href="{% static 'css/blog.css' %}">
    </head>
    <body>
        <div>
            <h1><a href="/">Django Girls Blog</a></h1>
        </div>

        {% for post in posts %}
            <div>
                <p>published: {{ post.published_date }}</p>
                <h1><a href="">{{ post.title }}</a></h1>
                <p>{{ post.text|linebreaksbr }}</p>
            </div>
        {% endfor %}
    </body>
</html>
```

De acuerdo, ¡guarda el archivo y actualiza el sitio!

¡Buen trabajo! ¿Tal vez nos gustaría también dar un poco de aire a nuestro sitio web y aumentar el margen en el lado izquierdo?. ¡Vamos a intentarlo!

blog/static/css/blog.css

```
body {
    padding-left: 15px;
}
```

Añade esto a tu CSS, guarda el archivo y ¡mira cómo funciona!

¿Quizá podríamos personalizar la tipografía del título? Pega esto en la sección `<head>` del archivo `blog/templates/blog/post_list.html` :

blog/templates/blog/post_list.html

```
<link href="//fonts.googleapis.com/css?family=Lobster&subset=latin,latin-ext" rel="stylesheet" type="text/css">
```

Como antes, revisa el orden y pon antes del enlace a `blog/static/css/blog.css` . Esta línea importará un estilo de letra llamada *Lobster* de Google Fonts (https://www.google.com/fonts).

Encuentra el bloque de declaración (el código entre las llaves `{` y `}`) `h1 a` en el archivo CSS `blog/static/css/blog.css`. Ahora añade la línea `font-family: 'Lobster';` entre las llaves y actualiza la página:

blog/static/css/blog.css

```
h1 a {
    color: #FCA205;
    font-family: 'Lobster';
}
```

¡Genial!

Como ya hemos dicho, CSS tiene un concepto de clases. Las clases te permiten dar un nombre a una parte del código HTML para aplicar estilos solo a esta parte, sin afectar a otras. ¡Esto puede ser súper útil! Quizá tienes dos divs haciendo algo diferente (como el encabezado y el texto de tu publicación). Las clases pueden ayudarte a asignarles estilos distintos.

Adelante, nombra algunas partes del código HTML. Añade una clase llamada `page-header` a tu `div` que contiene el encabezado, así:

blog/templates/blog/post_list.html

```
<div class="page-header">
    <h1><a href="/">Django Girls Blog</a></h1>
</div>
```

Y ahora añade una clase `post` a tu `div` que contiene una publicación del blog.

blog/templates/blog/post_list.html

```
<div class="post">
    <p>published: {{ post.published_date }}</p>
    <h1><a href="">{{ post.title }}</a></h1>
    <p>{{ post.text|linebreaksbr }}</p>
</div>
```

Ahora añadiremos bloques de declaración a varios selectores. Los selectores que comienzan con `.` hacen referencia a clases. Hay muchos tutoriales y explicaciones excelentes sobre CSS en la Web que te pueden ayudar a entender el código que sigue a continuación. Por ahora, copia y pega lo siguiente en tu archivo `blog/static/css/blog.css`:

blog/static/css/blog.css

```
.page-header {
    background-color: #ff9400;
    margin-top: 0;
    padding: 20px 20px 20px 40px;
}

.page-header h1, .page-header h1 a, .page-header h1 a:visited, .page-header h1 a:active {
    color: #ffffff;
    font-size: 36pt;
    text-decoration: none;
}

.content {
    margin-left: 40px;
}

h1, h2, h3, h4 {
    font-family: 'Lobster', cursive;
}

.date {
    float: right;
    color: #828282;
}

.save {
    float: right;
}

.post-form textarea, .post-form input {
    width: 100%;
}

.top-menu, .top-menu:hover, .top-menu:visited {
    color: #ffffff;
    float: right;
    font-size: 26pt;
    margin-right: 20px;
}

.post {
    margin-bottom: 70px;
}

.post h1 a, .post h1 a:visited {
    color: #000000;
}
```

Luego rodea el código HTML que muestra los posts con declaraciones de clases. Cambia esto:

blog/templates/blog/post_list.html

```
{% for post in posts %}
    <div class="post">
        <p>published: {{ post.published_date }}</p>
        <h1><a href="">{{ post.title }}</a></h1>
        <p>{{ post.text|linebreaksbr }}</p>
    </div>
{% endfor %}
```

en `blog/templates/blog/post_list.html` por esto:

blog/templates/blog/post_list.html

```
<div class="content container">
    <div class="row">
        <div class="col-md-8">
            {% for post in posts %}
                <div class="post">
                    <div class="date">
                        <p>published: {{ post.published_date }}</p>
                    </div>
                    <h1><a href="">{{ post.title }}</a></h1>
                    <p>{{ post.text|linebreaksbr }}</p>
                </div>
            {% endfor %}
        </div>
    </div>
</div>
```

Guarda estos archivos y recarga tu sitio.

¡Woohoo! Queda genial, ¿Verdad? Mira el código que acabamos de pegar para encontrar los lugares donde añadimos clases en el HTML y las usamos en el CSS. ¿Qué cambiarías si quisieras que la fecha fuera color turquesa?

No tengas miedo a experimentar con este CSS un poco y tratar de cambiar algunas cosas. Jugar con el CSS te puede ayudar a entender lo que hacen las distintas secciones. Si algo deja de funcionar, no te preocupes, ¡siempre puedes deshacerlo!

Realmente te recomendamos seguir este Curso de HTML y CSS de CodeAcademy. Puede ayudarte a aprender todo lo que necesitas para hacer tus websites más bonitos con CSS.

¡¿Lista para el siguiente capítulo?! :)

Extendiendo plantillas

Otra cosa buena que tiene Django es la **extensión de plantillas**. ¿Qué significa? Significa que puedes reusar partes del HTML para diferentes páginas del sitio web.

Las plantillas son útiles cuando quieres utilizar la misma información o el mismo diseño en más de un lugar. No tienes que repetirte a ti misma en cada archivo. Y si quieres cambiar algo, no tienes que hacerlo en cada plantilla, sólo en una!

Crea una plantilla base

Una plantilla base es la plantilla más básica que extiendes en cada página de tu sitio web.

Vamos a crear un archivo `base.html` en `blog/templates/blog/` :

```
blog
└──templates
    └──blog
            base.html
            post_list.html
```

Ahora, ábrelo en el editor de código y copia todo el contenido de `post_list.html` en `base.html` , así:

blog/templates/blog/base.html

```
{% load static %}
<html>
    <head>
        <title>Django Girls blog</title>
        <link rel="stylesheet" href="//maxcdn.bootstrapcdn.com/bootstrap/3.2.0/css/bootstrap.min.css">
        <link rel="stylesheet" href="//maxcdn.bootstrapcdn.com/bootstrap/3.2.0/css/bootstrap-theme.min.css">
        <link href='//fonts.googleapis.com/css?family=Lobster&subset=latin,latin-ext' rel='stylesheet' type='text/css'
>
        <link rel="stylesheet" href="{% static 'css/blog.css' %}">
    </head>
    <body>
        <div class="page-header">
            <h1><a href="/">Django Girls Blog</a></h1>
        </div>

        <div class="content container">
            <div class="row">
                <div class="col-md-8">
                {% for post in posts %}
                    <div class="post">
                        <div class="date">
                            {{ post.published_date }}
                        </div>
                        <h1><a href="">{{ post.title }}</a></h1>
                        <p>{{ post.text|linebreaksbr }}</p>
                    </div>
                {% endfor %}
                </div>
            </div>
        </div>
    </body>
</html>
```

Luego, en `base.html` reemplaza por completo tu `<body>` (todo lo que haya entre `<body>` and `</body>`) con esto:

blog/templates/blog/base.html

```
<body>
    <div class="page-header">
        <h1><a href="/">Django Girls Blog</a></h1>
    </div>
    <div class="content container">
        <div class="row">
            <div class="col-md-8">
            {% block content %}
            {% endblock %}
            </div>
        </div>
    </div>
</body>
```

Seguro que ya te has dado cuenta de que lo que hemos hecho ha sido cambiar todo lo que había entre `{% for post in posts %}` y `{% endfor %}` por

blog/templates/blog/base.html

```
{% block content %}
{% endblock %}
```

Pero ¿por qué? ¡Acabas de crear un bloque! Hemos usado la etiqueta de plantilla `{% block %}` para crear un área en la que se insertará HTML. Ese HTML vendrá de otra plantilla que extiende esta (`base.html`). Enseguida te enseñamos cómo se hace.

Ahora guarda `base.html` y abre `blog/templates/blog/post_list.html` de nuevo en el editor. Quita todo lo que hay encima de `{% for post in posts %}` y por debajo de `{% endfor %}`. Cuando termines, el archivo tendrá este aspecto:

blog/templates/blog/post_list.html

```
{% for post in posts %}
    <div class="post">
        <div class="date">
            {{ post.published_date }}
        </div>
        <h1><a href="">{{ post.title }}</a></h1>
        <p>{{ post.text|linebreaksbr }}</p>
    </div>
{% endfor %}
```

Queremos utilizar esto como parte de nuestra plantilla en los bloques de contenido. ¡Es hora de añadir etiquetas de bloque en este archivo!

Tu etiqueta de bloque debe ser la misma que la etiqueta del archivo `base.html`. También querrás que incluya todo el código que va en los bloques de contenido. Para ello, pon todo entre `{% block content %}` y `{% endblock %}`. Algo como esto:

blog/templates/blog/post_list.html

```
{% block content %}
    {% for post in posts %}
        <div class="post">
            <div class="date">
                {{ post.published_date }}
            </div>
            <h1><a href="">{{ post.title }}</a></h1>
            <p>{{ post.text|linebreaksbr }}</p>
        </div>
    {% endfor %}
{% endblock %}
```

Solo falta una cosa. Tenemos que conectar estas dos plantillas. ¡Esto es lo que significa extender plantillas! Para eso tenemos que añadir una etiqueta "extends" al comienzo del archivo. Así:

blog/templates/blog/post_list.html

```
{% extends 'blog/base.html' %}

{% block content %}
    {% for post in posts %}
        <div class="post">
            <div class="date">
                {{ post.published_date }}
            </div>
            <h1><a href="">{{ post.title }}</a></h1>
            <p>{{ post.text|linebreaksbr }}</p>
        </div>
    {% endfor %}
{% endblock %}
```

¡Y ya está! Guarda el fichero y comprueba que el sitio web sigue funcionando como antes. :)

> Si te sale el error `TemplateDoesNotExist`, que significa que no hay ningún archivo `blog/base.html` y tienes `runserver` corriendo en la consola. Intenta pararlo, pulsando Ctrl+C (teclas Control y C a la vez) en la consola y reiniciarlo con el comando `python manage.py runserver`.

Extiende tu aplicación

Ya hemos completado todos los pasos necesarios para la creación de nuestro sitio web: sabemos cómo escribir un model, url, view y template. También sabemos cómo hacer que nuestro sitio web tenga buen aspecto.

¡Hora de practicar!

Lo primero que necesitamos en nuestro blog es, obviamente, una página para mostrar un post, ¿cierto?

Ya tenemos un modelo `Post`, así que no necesitamos añadir nada a `models.py`.

Crea un enlace a la página de detalle de una publicación

Empezaremos añadiendo un enlace al fichero `blog/templates/blog/post_list.html`. Ábrelo en el editor; de momento debería tener este contenido:

blog/templates/blog/post_list.html

```
{% extends 'blog/base.html' %}

{% block content %}
    {% for post in posts %}
        <div class="post">
            <div class="date">
                {{ post.published_date }}
            </div>
            <h1><a href="">{{ post.title }}</a></h1>
            <p>{{ post.text|linebreaksbr }}</p>
        </div>
    {% endfor %}
{% endblock %}
```

Queremos tener un link del titulo de una publicación en la lista de publicaciones al detalle de la misma. Vamos a cambiar `<h1><a href="">{{ post.title }}</a></h1>` para que se enlace a la página de detalles de publicación:

blog/templates/blog/post_list.html

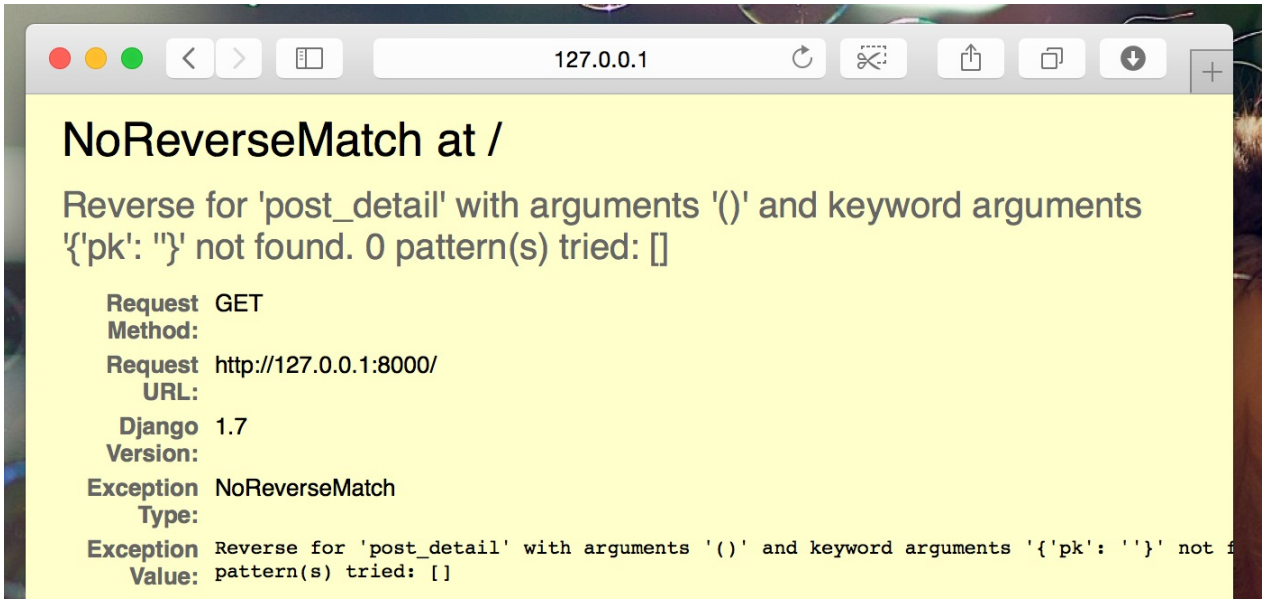
```
<h1><a href="{% url 'post_detail' pk=post.pk %}">{{ post.title }}</a></h1>
```

Es hora de explicar el misterioso `{% url 'post_detail' pk=post.pk %}`. Como probablemente sospeches, la notación `{% %}` significa que estamos utilizando Django template tags. ¡Esta vez usaremos uno que creará un URL para nosotros!

La parte de `post_detail` significa que Django estará esperando un URL en `blog/urls.py` con el nombre=post_detail

¿Y ahora qué pasa con < 0>pk=post.pk</code>? `pk` se refiere a primary key (clave primaria), la cual es un nombre único por cada registro en una base de datos. Debido a que no especificamos una llave primaria en nuestro modelo `Post`, Django creará una por nosotros (por defecto, un número que incrementa una unidad por cada registro, por ejemplo, 1, 2, 3) y lo añadirá como un campo llamado `pk` a cada uno de nuestros posts. Accedemos a la clave primaria escribiendo `post.pk`, del mismo modo en que accedemos a otros campos (`título`, `autor`, etc.) en nuestro objeto `Post`!

Ahora cuando vayamos a: http://127.0.0.1:8000/ tendremos un error (como era de esperar, ya que no tenemos una URL o una *vista* para `post_detail`). Se verá así:



Crea una URL al detalle de una publicación

Vamos a crear una URL en `urls.py` para nuestra *view* `post_detail`!

Queremos que el detalle de la primera entrada se visualice en esta **URL**: http://127.0.0.1:8000/post/1/

Vamos a crear una URL en el fichero `blog/urls.py` que dirija a Django hacia una *vista* llamada `post_detail`, que mostrará una entrada de blog completa. Abre el fichero `blog/urls.py` en el editor, y añade la línea `path('post/<int:pk>)/', views.post_detail, name='post_detail'),` para que el fichero quede así:

blog/urls.py

```
from django.urls import path
from . import views

urlpatterns = [
    path('', views.post_list, name='post_list'),
    path('post/<int:pk>/', views.post_detail, name='post_detail'),
]
```

Esta parte `post/<int:pk>/` especifica un patrón de URL – ahora lo explicamos:

- `post/` significa que la URL debería empezar con la palabra **post** seguida por una */*. Hasta aquí bien.
- `<int:pk>` – esta parte tiene más miga. Significa que Django buscará un número entero y se lo pasará a la vista en una variable llamada `pk`.
- `/` – necesitamos otra */* al final de la URL.

Esto quiere decir que si pones `http://127.0.0.1:8000/post/5/` en tu navegador, Django entenderá que estás buscando una *vista* llamada `post_detail` y transferirá la información de que `pk` es igual a `5` a esa *vista*.

OK, hemos añadido un nuevo patrón de URL a `blog/urls.py`! Actualizamos la pagina: http://127.0.0.1:8000/ y boom! El servidor vuelve a dejar de funcionar. Echa un vistazo a la consola – como era de esperar, hay otro error!

```
    return _bootstrap._gcd_import(name[level:], package, level)
  File "<frozen importlib._bootstrap>", line 2231, in _gcd_import
  File "<frozen importlib._bootstrap>", line 2214, in _find_and_load
  File "<frozen importlib._bootstrap>", line 2203, in _find_and_load_unlocked
  File "<frozen importlib._bootstrap>", line 1200, in _load_unlocked
  File "<frozen importlib._bootstrap>", line 1129, in _exec
  File "<frozen importlib._bootstrap>", line 1448, in exec_module
  File "<frozen importlib._bootstrap>", line 321, in _call_with_frames_removed
  File "/home/hel/code/djangogirls/workthrough/blog/urls.py", line 6, in <module>
    url(r'^post/(?P<pk>[0-9]+)/$', views.post_detail, name='post_detail'),
AttributeError: 'module' object has no attribute 'post_detail'
```

¿Recuerdas cual es el próximo paso? ¡Añadir una vista!

Añade la vista de detalle de la publicación

Esta vez nuestra *vista* tomará un parámetro adicional `pk`. Nuestra *vista* necesita recibirlo, ¿verdad? Así que definiremos nuestra función como `def post_detail (request, pk):`. Ten en cuenta que tenemos que usar exactamente el mismo nombre que especificamos en las urls (`pk`). ¡Omitir esta variable es incorrecto y resultará en un error!

Ahora, queremos obtener solo un post. Para ello podemos usar querysets como este:

blog/views.py

```
Post.objects.get(pk=pk)
```

Pero este código tiene un problema. Si no hay ningún `Post` con esa `clave primaria` (`pk`), ¡tendremos un error muy feo!

¡No queremos eso! Por suerte, Django tiene una función que se encarga de eso: `get_object_or_404`. En caso de que no haya ningún `Post` con el `pk` dado se mostrará una página mucho más agradable, `Page Not Found 404`.

La buena noticia es que puedes crear tu propia página `Page Not Found` y diseñarla como desees. Pero por ahora no es tan importante, así que lo omitiremos.

¡Es hora de agregar una *view* a nuestro archivo `views.py` !

En `blog/urls.py` creamos un regla de URL denominada `post_detail` que hace referencia a una vista llamada `view.post_detail` . Esto significa que Django va a estar esperando una función llamada `post_detail` de vista en `blog/views.py` .

Deberíamos abrir `blog/views.py` en el editor y añadir el siguiente código cerca de los otros import `from` :

blog/views.py

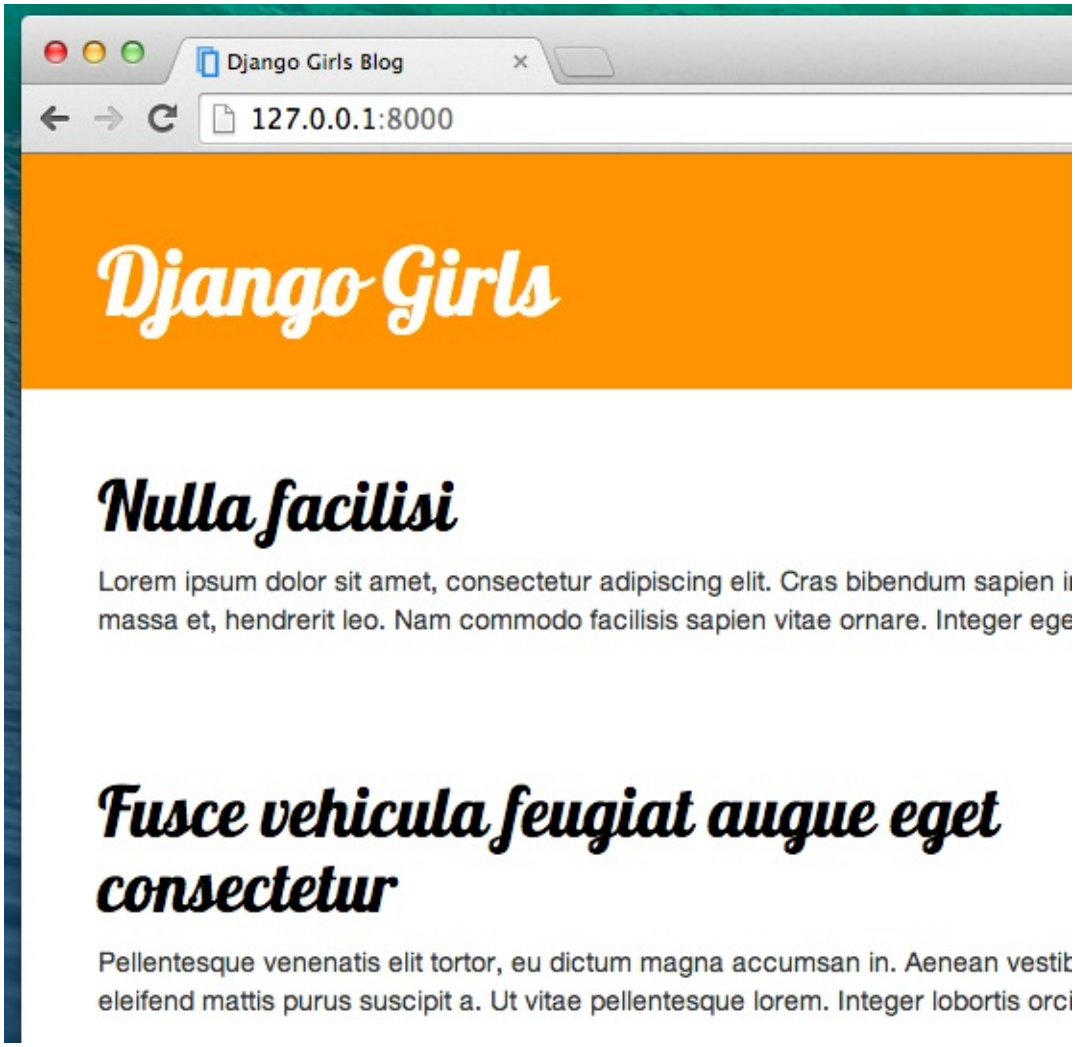
```
from django.shortcuts import render, get_object_or_404
```

Y al final del archivo agregamos nuestra *view*:

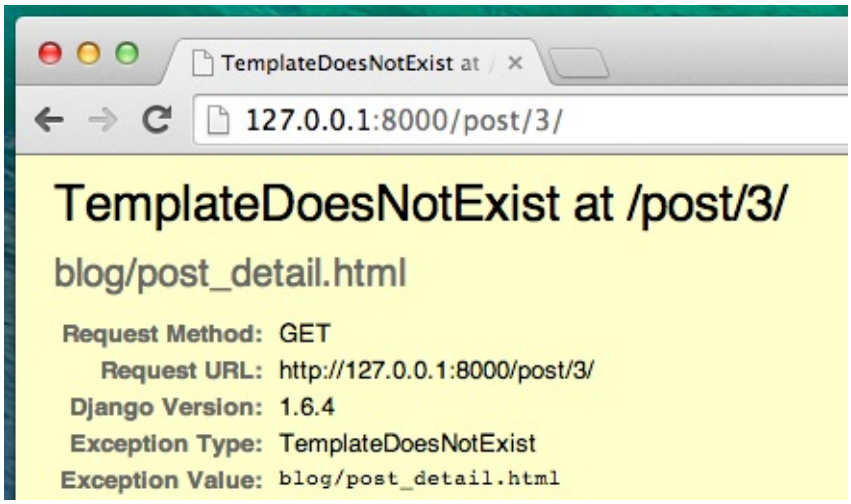
blog/views.py

```
def post_detail(request, pk):
    post = get_object_or_404(Post, pk=pk)
    return render(request, 'blog/post_detail.html', {'post': post})
```

Sí. Es hora de actualizar la página: http://127.0.0.1:8000/



¡Funcionó! Pero ¿qué pasa cuando haces click en un enlace en el título del post?



¡Oh no! ¡Otro error! Pero ya sabemos cómo lidiar con eso, ¿no? ¡Tenemos que añadir una plantilla!

Crear una plantilla para post detail

Vamos crear un fichero en `blog/templates/blog` llamado `post_detail.html`, y abrirlo en el editor de código.

Se verá así:

blog/templates/blog/post_detail.html

```
{% extends 'blog/base.html' %}

{% block content %}
    <div class="post">
        {% if post.published_date %}
            <div class="date">
                {{ post.published_date }}
            </div>
        {% endif %}
        <h1>{{ post.title }}</h1>
        <p>{{ post.text|linebreaksbr }}</p>
    </div>
{% endblock %}
```

Una vez más estamos extendiendo `base.html`. En el bloque `content` queremos mostrar la fecha de publicación (si existe), título y texto de nuestros posts. Pero deberíamos discutir algunas cosas importantes, ¿cierto?

`{% if ... %} ... {% endif %}` es un template tag que podemos usar cuando querramos ver algo. (Recuerdas `if ... else ..` del capítulo **Intruducción a Python**?) Ahora queremos mirar si la `published_date` de un post no esta vacía.

Bien, podemos actualizar nuestra página y ver si `TemplateDoesNotExist` se ha ido.

¡Yay! ¡Funciona!

Hora de despliegue!

Sería bueno verificar que tu sitio web aún funcionará en PythonAnywhere, ¿cierto? Intentemos desplegar de nuevo.

command-line

```
$ git status
$ git add -A .
$ git status
$ git commit -m "Agregadas vistas y plantilla para el detalle del post del blog así como también CSS para el sitio."
$ git push
```

Luego, en una consola Bash de PythonAnywhere:

command-line

```
$ cd ~/<your-pythonanywhere-username>.pythonanywhere.com
$ git pull
[...]
```

(Recuerda sustituir `<your-pythonanywhere-username>` con tu nombre de PythonAnywhere real, sin los signos de ángulo).

Actualizar los ficheros estáticos (static files) en el servidor

Normalmente, los servidores como PythonAnywhere tratan los ficheros estáticos (como los ficheros CSS) de manera diferente a los ficheros de Python. Se llaman estáticos porque el servidor no debe ejecutarlos, sino servirlos tal cual. Y por ello se tratan por separado para servirlos más rápido. Como consecuencia, si cambiamos nuestros ficheros CSS, tenemos que ejecutar un comando extra en el servidor para decirle que los actualice. Este comando se llama `collectstatic`.

Activa el virtualenv si no estaba activado de antes (en PythonAnywhere se usa el comando `workon`, es igual que el comando `source myenv/bin/activate` que usamos en local):

command-line

```
$ workon <your-pythonanywhere-username>.pythonanywhere.com
(ola.pythonanywhere.com)$ python manage.py collectstatic
[...]
```

El comando `manage.py collectstatic` es un poco como el comando `manage.py migrate`. Hacemos cambios en nuestro código y luego le decimos a Django que los *aplique*, bien a la colección de ficheros estáticos o bien a la base de datos.

En cualquier caso, ya podemos ir a la página "Web" (botón en la esquina superior derecha de la consola), hacer click en **Reload**, y mirar la página https://yourname.pythonanywhere.com para ver el resultado.

¡Y eso debería ser todo! Felicidades :)

Formularios de Django

Lo último que haremos en nuestro sitio web será crear una forma agradable de agregar y editar posts en el blog. El `admin` de Django está bien, pero es bastante difícil de personalizar y hacerlo bonito. Con `forms` tendremos un poder absoluto sobre nuestra interfaz; ¡podemos hacer casi cualquier cosa que podamos imaginar!

Lo bueno de los formularios de Django es que podemos definirlos desde cero o crear un `ModelForm`, el cual guardará el resultado del formulario en el modelo.

Esto es exactamente lo que queremos hacer: crearemos un formulario para nuestro modelo `Post`.

Como cada parte importante de Django, los formularios tienen su propio archivo: `forms.py`.

Necesitamos crear un archivo con este nombre en el directorio `blog`.

```
blog
   └── forms.py
```

Vale, ábrelo en el editor de código y teclea lo siguiente:

blog/forms.py

```
from django import forms

from .models import Post

class PostForm(forms.ModelForm):

    class Meta:
        model = Post
        fields = ('title', 'text',)
```

Lo primero, necesitamos importar Django forms (`from django import forms`) y nuestro modelo `Post` (`from .models import Post`).

`PostForm`, como probablemente sospechas, es el nombre de nuestro formulario. Necesitamos decirle a Django que este formulario es un `ModelForm` (así Django hará algo de magia por nosotros) - `forms.ModelForm` es responsable de ello.

Luego, tenemos `class Meta`, donde le decimos a Django que modelo debe ser utilizado para crear este formulario (`model = Post`).

Finalmente, podemos decir que campo(s) deberían estar en nuestro formulario. En este escenario sólo queremos `title` y `text` para ser mostrados - `author` será la persona que esta conectada (¡tú!) y `created_date` se definirá automáticamente cuando creemos un post (es decir, en el código), ¿cierto?

¡Y eso es todo! Todo lo que necesitamos hacer ahora es usar el formulario en una *view* y mostrarla en una plantilla.

Una vez más vamos a crear: un enlace a la página, una dirección URL, una vista y una plantilla.

Enlace a una página con el formulario

Ahora toca abrir el fichero `blog/templates/blog/base.html` en el editor. Vamos a añadir un enlace en el `div` llamado `page-header`:

blog/templates/blog/post_base.html

```
<a href="{% url 'post_new' %}" class="top-menu"><span class="glyphicon glyphicon-plus"></span></a>
```

Ten en cuenta que queremos llamar a nuestra nueva vista `post_new`. La clase `"glyphicon glyphicon-plus"` es proporcionada por el tema de bootstrap que estamos utilizando, y nos mostrara un signo de suma.

Después de agregar la línea, tu archivo html debería lucir de esta forma:

blog/templates/blog/post_base.html

```
{% load static %}
<html>
    <head>
        <title>Django Girls blog</title>
        <link rel="stylesheet" href="//maxcdn.bootstrapcdn.com/bootstrap/3.2.0/css/bootstrap.min.css">
        <link rel="stylesheet" href="//maxcdn.bootstrapcdn.com/bootstrap/3.2.0/css/bootstrap-theme.min.css">
        <link href='//fonts.googleapis.com/css?family=Lobster&subset=latin,latin-ext' rel='stylesheet' type='text/css'
>
        <link rel="stylesheet" href="{% static 'css/blog.css' %}">
    </head>
    <body>
        <div class="page-header">
            <a href="{% url 'post_new' %}" class="top-menu"><span class="glyphicon glyphicon-plus"></span></a>
            <h1><a href="/">Django Girls Blog</a></h1>
        </div>
        <div class="content container">
            <div class="row">
                <div class="col-md-8">
                    {% block content %}
                    {% endblock %}
                </div>
            </div>
        </div>
    </body>
</html>
```

Después de guardar y refrescar la página http://127.0.0.1:8000 verás el - ya conocido - error `NoReverseMatch`. ¿Es así? ¡Vamos bien!

URL

Abrimos `blog/urls.py` en el editor para añadir una línea:

blog/urls.py

```
path('post/new', views.post_new, name='post_new'),
```

Y el código final tendrá este aspecto:

blog/urls.py

```
from django.urls import path
from . import views

urlpatterns = [
    path('', views.post_list, name='post_list'),
    path('post/<int:pk>/', views.post_detail, name='post_detail'),
    path('post/new/', views.post_new, name='post_new'),
]
```

Después de actualizar el sitio, veremos un `AttributeError`, puesto que no tenemos la vista `post_new` implementada. Añadamosla de una vez.

Vista post_new

Ahora abre el fichero `blog/views.py` en el editor y añade estas líneas con el resto de imports `from` :

blog/views.py

```
from .forms import PostForm
```

Y ahora nuestra *vista*:

blog/views.py

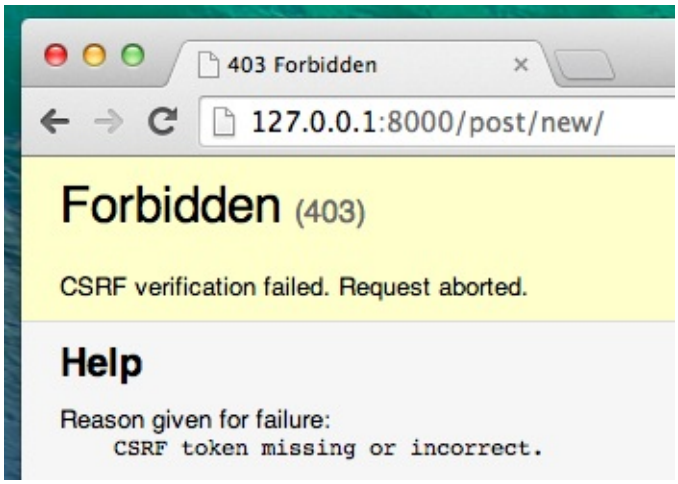
```
def post_new(request):
    form = PostForm()
    return render(request, 'blog/post_edit.html', {'form': form})
```

Para crear un nuevo formulario `Post` , tenemos que llamar a `PostForm()` y pasarlo a la plantilla. Volveremos a esta *vista* pero, por ahora, vamos a crear rápidamente una plantilla para el formulario.

Plantilla

Tenemos que crear un fichero `post_edit.html` el el directorio `blog/templates/blog` , y abrirlo en el editor de código. Para hacer que un formulario funcione necesitamos varias cosas:

- Tenemos que mostrar el formulario. Podemos hacerlo, por ejemplo, con un sencillo `{{ form.as_p }}` .
- La línea anterior tiene que estar dentro de una etiqueta de formuLario HTML: `<form method="POST">...</form>` .
- Necesitamos un botón `Guardar` . Lo hacemos con un botón HTML: `<button type='submit'>Save</button>` .
- Finalmente justo después de abrir la etiqueta `<form ...>` tenemos que añadir `{% csrf_token %}` . ¡Esto es muy importante ya que hace que tus formularios sean seguros! Si olvidas este pedazo, Django se molestará cuando intentes guardar el formulario:



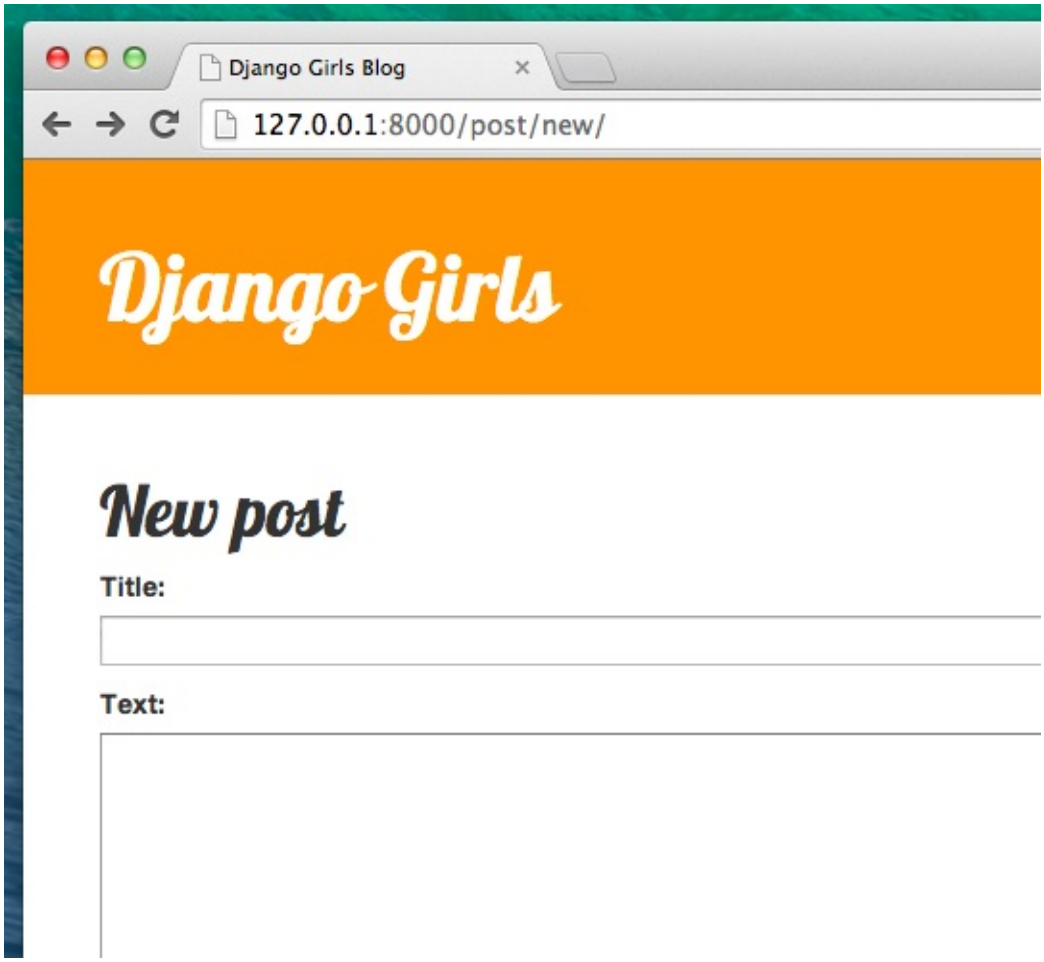
Bueno, miremos se debería ver el HTML en `post_edit.html` :

blog/templates/blog/post_edit.html

```
{% extends 'blog/base.html' %}

{% block content %}
    <h1>New post</h1>
    <form method="POST" class="post-form">{% csrf_token %}
        {{ form.as_p }}
        <button type="submit" class="save btn btn-default">Save</button>
    </form>
{% endblock %}
```

¡Es hora de actualizar! ¡Si! ¡Tu formulario se muestra!



Pero, ¡un momento! Si escribes algo en los campos `title` y `text` y tratas de guardar los cambios - ¿qué pasará?

¡Nada! Una vez más estamos en la misma página y el texto se ha ido... no se añade ningún post nuevo. Entonces, ¿qué ha ido mal?

La respuesta es: nada. Tenemos que trabajar un poco más en nuestra *vista*.

Guardar el formulario

Abre `blog/views.py` de nuevo en el editor. De momento todo lo que tenemos en la vista `post_new` es lo siguiente:

blog/views.py

```
def post_new(request):
    form = PostForm()
    return render(request, 'blog/post_edit.html', {'form': form})
```

Cuando enviamos el formulario somos redirigidos a la misma vista, pero esta vez tenemos algunos datos adicionales en `request`, más específicamente en `request.POST` (el nombre no tiene nada que ver con un post del blog, se refiere a que estamos "publicando" -en inglés, posting- datos). ¿Recuerdas que en el archivo HTML la definición de `<form>` tenía la variable `method="POST"` ? Todos los campos del formulario estan ahora en `request.POST`. No deberías renombrar la variable `POST` (el único nombre que también es válido para la variable `method` es `GET`, pero no tenemos tiempo para explicar cuál es la diferencia).

En nuestra *vista* tenemos dos situaciones distintas que manejar: primero, cuando accedemos a la pagina por primera vez y queremos un formulario vacío, y segundo, cuando regresamos a la *vista* con los datos del formulario que acabamos de ingresar. Así que tenemos que añadir una condición (utilizaremos `if` para eso):

blog/views.py

```
if request.method == "POST":
    [...]
else:
    form = PostForm()
```

Es hora de rellenar los puntos `[...]`. Si el `método` es `POST` entonces querremos construir el `PostForm` con datos del formulario, cierto? Lo haremos de la siguiente manera:

blog/views.py

```
form = PostForm(request.POST)
```

Lo siguiente es verificar si el formulario está correcto (si todos los campos necesarios están definidos y no hay valores incorrectos). Lo hacemos con `form.is_valid()`.

Comprobamos que el formulario es válido y, si es así, ¡lo podemos salvar!

blog/views.py

```
if form.is_valid():
    post = form.save(commit=False)
    post.author = request.user
    post.published_date = timezone.now()
    post.save()
```

Básicamente, tenemos que hacer dos cosas: guardamos el formulario con `form.save` y añadimos un autor (ya que no había ningún campo de `author` en el `PostForm` y este campo es obligatorio). `commit=False` significa que no queremos guardar el modelo `Post` aún - queremos añadir el autor primero. La mayoría de las veces utilizarás `form.save()`, sin `commit=False`, pero en este caso, tenemos que hacerlo. `post.save()` conservará los cambios (añadiendo a autor) y se creará una nuevo post en el blog!

Por último, sería genial si podemos inmediatamente ir a la página `post_detail` del nuevo post, ¿no? Para hacerlo necesitamos importar algo más:

blog/views.py

```
from django.shortcuts import redirect
```

Agrégalo al principio del archivo. Y ahora podemos decir: "vé a la página `post_detail` del post recién creado":

blog/views.py

```
return redirect('post_detail', pk=post.pk)
```

`post_detail` es el nombre de la vista a la que queremos ir. ¿Recuerdas que esta *view* requiere una variable `pk`? Para pasarlo a las vistas utilizamos `pk=post.pk`, donde `post` es el post recién creado!

Bien, hablamos mucho, pero probablemente queremos ver como se ve la *vista*, ¿verdad?

blog/views.py

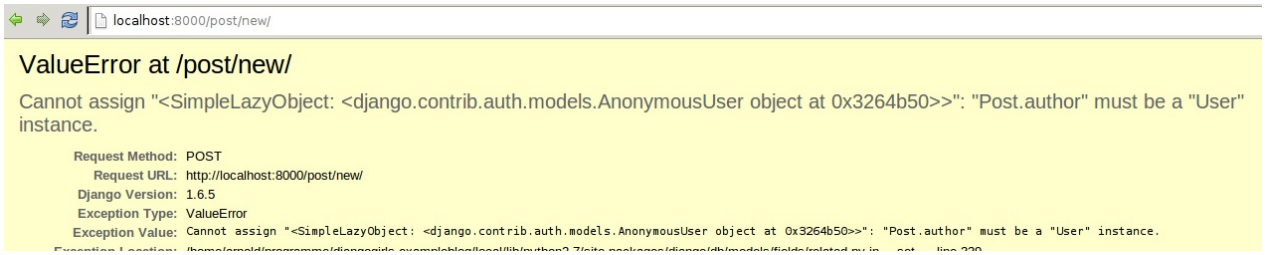
```
def post_new(request):
    if request.method == "POST":
        form = PostForm(request.POST)
        if form.is_valid():
            post = form.save(commit=False)
            post.author = request.user
            post.published_date = timezone.now()
            post.save()
            return redirect('blog.views.post_detail', pk=post.pk)
    else:
        form = PostForm()
    return render(request, 'blog/post_edit.html', {'form': form})
```

Veamos si funciona. Ve a la página http://127.0.0.1:8000/post/new/, añade un `title` y un `text`, guárdalo... y voilà! Se ha añadido el nuevo post al blog y somos redirigidos a la página de `post_detail`!

Puede que hayas notado que estamos indicando la fecha de publicación antes de guardar el post. Más adelante introduciremos un *botón de publicar* en el libro **Django Girls Tutorial: Extensions**.

¡Eso es genial!

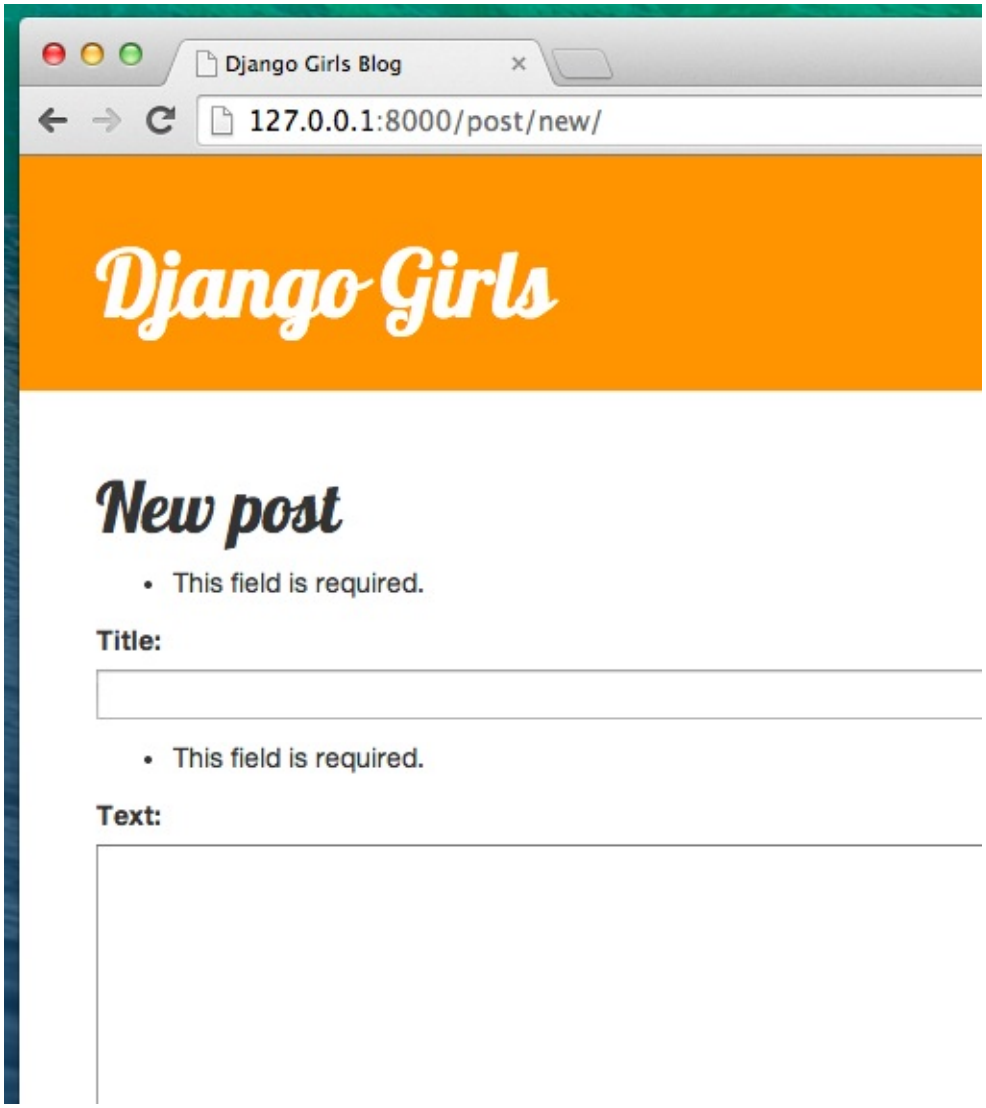
> Como recientemente hemos utilizado la interfaz de administrador de Django, el sistema piensa que estamos conectados. Hay algunas situaciones que podrían llevarnos a desconectarnos (cerrando el navegador, reiniciando la base de datos, etc.). Si estás recibiendo errores al crear un post que indican la falta de inicio de sesión de usuario, dirígete a la pagina de admin http://127.0.0.1:8000/admin e inicia sesión nuevamente. Esto resolverá el problema temporalmente. Hay un arreglo permanente esperándote en el capítulo **Tarea: ¡Añadir seguridad a tu sitio web!** después del tutorial principal.



Validación de formularios

Ahora, vamos a enseñarte qué tan bueno es Django forms. Un post del blog debe tener los campos `title` y `text`. En nuestro modelo `Post` no dijimos (a diferencia de `published_date`) que estos campos no son requeridos, así que Django, por defecto, espera que estén definidos.

Trata de guardar el formulario sin `title` y `text`. ¡Adivina qué pasará!



Django se encarga de validar que todos los campos en el formulario estén correctos. ¿No es genial?

Editar formulario

Ahora sabemos cómo agregar un nuevo formulario. Pero, ¿qué pasa si queremos editar uno existente? Es muy similar a lo que acabamos de hacer. Creemos rápidamente algunas cosas importantes. (si no entiendes algo, pregúntale a tu tutora o tutor, o revisa lo capítulos anteriores, son temas que ya hemos cubierto.)

Abre `blog/templates/blog/post_detail.html` en el editor de código y añade la línea

blog/templates/blog/post_detail.html

```
<a class="btn btn-default" href="{% url 'post_edit' pk=post.pk %}"><span class="glyphicon glyphicon-pencil"></span></a
>
```

para que la plantilla quede así:

blog/templates/blog/post_detail.html

```
{% extends 'blog/base.html' %}

{% block content %}
    <div class="post">
        {% if post.published_date %}
            <div class="date">
                {{ post.published_date }}
            </div>
        {% endif %}
        <a class="btn btn-default" href="{% url 'post_edit' pk=post.pk %}"><span class="glyphicon glyphicon-pencil"></
span></a>
        <h1>{{ post.title }}</h1>
        <p>{{ post.text|linebreaksbr }}</p>
    </div>
{% endblock %}
```

Abre `blog/urls.py` en el editor y añade esta línea:

blog/urls.py

```
    path('post/<int:pk>/edit/', views.post_edit, name='post_edit'),
```

Vamos a reusar la plantilla `blog/templates/blog/post_edit.html`, así que lo último que nos falta es una *view*.

Abre `blog/views.py` en el editor de código y añade esto al final del todo:

blog/views.py

```
def post_edit(request, pk):
    post = get_object_or_404(Post, pk=pk)
    if request.method == "POST":
        form = PostForm(request.POST, instance=post)
        if form.is_valid():
            post = form.save(commit=False)
            post.author = request.user
            post.published_date = timezone.now()
            post.save()
            return redirect('post_detail', pk=post.pk)
    else:
        form = PostForm(instance=post)
    return render(request, 'blog/post_edit.html', {'form': form})
```

Esto se ve casi exactamente igual a nuestra view `post_new`, ¿no? Pero no del todo. Primero: pasamos un parámetro extra `pk` de los urls. Segundo: obtenemos el modelo `Post` que queremos editar con `get_object_or_404(Post, pk=pk)` y después, al crear el formulario pasamos este post como una `instancia` tanto al guardar el formulario…

blog/views.py

```
form = PostForm(request.POST, instance=post)
```

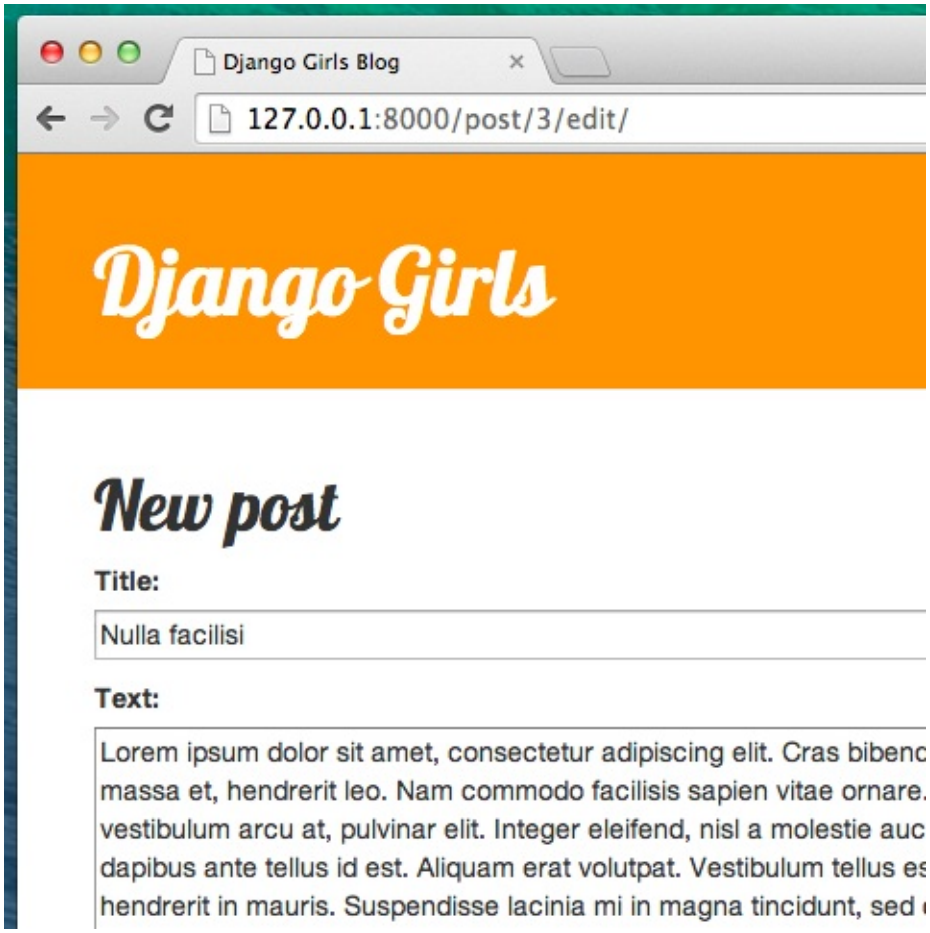
... y justo cuando abrimos un formulario con este post para editarlo:

blog/views.py

```
form = PostForm(instance=post)
```

Ok, ¡vamos a probar si funciona! Dirígete a la página `post_detail`. Ahí debe haber un botón para editar en la esquina superior derecha:

Al dar click ahí, debes ver el formulario con nuestro post del blog:



¡Siéntete libre de cambiar el título o el texto y guarda los cambios!

¡Felicitaciones! ¡Tu aplicación está cada vez más completa!

Si quieres saber más sobre los Django forms, puedes encontrar la documentación aquí: https://docs.djangoproject.com/en/2.0/topics/forms/

Seguridad

¡Poder crear nuevas publicaciones haciendo click en un enlace es genial! Pero, ahora mismo, cualquiera que visite tu página podría publicar un nuevo post y seguro que eso no es lo que quieres. Vamos a hacer que el botón sea visible para ti pero no para nadie más.

Abre `blog/templates/blog/base.html` en el editor, busca el `div` `page-header` y la etiqueta del enlace (anchor) que pusimos antes. Debería ser algo así:

blog/templates/blog/base.html

```
<a href="{% url 'post_new' %}" class="top-menu"><span class="glyphicon glyphicon-plus"></span></a>
```

Vamos a añadir otra etiqueta `{% if %}` que hará que el enlace sólo parezca para los usuarios que hayan iniciado sesión en el admin. Ahora mismo, ¡eres sólo tú! Cambia la etiqueta `<a>` para que se parezca a esto:

blog/templates/blog/base.html

```
{% if user.is_authenticated %}
    <a href="{% url 'post_new' %}" class="top-menu"><span class="glyphicon glyphicon-plus"></span></a>
{% endif %}
```

Este `{% if %}` hará que el enlace sólo se envíe al navegador si el usuario que solicita la página ha iniciado sesión. Esto no protege completamente la creación de nuevas entradas, pero es un buen primer paso. Veremos más sobre seguridad en el libro de extensiones.

Recuerdas el icono de "editar" que acabamos de añadir a nuestra pagina de detalles? También queremos añadir lo mismo aquí, así otras personas no podrán editar posts existentes.

Abre `blog/templates/blog/post_detail.html` en el editor y busca esta línea:

blog/templates/blog/post_detail.html

```
<a class="btn btn-default" href="{% url 'post_edit' pk=post.pk %}"><span class="glyphicon glyphicon-pencil"></span></a
>
```

Cámbiala a lo siguiente:

blog/templates/blog/post_detail.html

```
{% if user.is_authenticated %}
    <a class="btn btn-default" href="{% url 'post_edit' pk=post.pk %}"><span class="glyphicon glyphicon-pencil"></span
></a>
{% endif %}
```

Dado que es probable que estés conectado, si actualizas la pagina, no veras nada diferente. Carga la pagina en un navegador diferente o en una ventana en modo incógnito ("privado" en Windows Edge) y verás que el link no aparece, y el icono tampoco!

Una cosa más: ¡Tiempo de despliegue!

Veamos si todo esto funciona en PythonAnywhere. ¡Tiempo de hacer otro despliegue!

- Lo primero, haz commit de tus últimos cambios y súbelo (push) a GitHub:

command-line

```
$ git status
$ git add --all .
$ git status
$ git commit -m "Added views to create/edit blog post inside the site."
$ git push
```

- Luego, en una consola Bash de PythonAnywhere

command-line

```
$ cd ~/<your-pythonanywhere-username>.pythonanywhere.com $ git pull
[...]
```

(Recuerda sustituir `<your-pythonanywhere-username>` con tu nombre de PythonAnywhere real, sin los signos de ángulo).

- Para terminar ve a "Web" page (usa el botón del menú de la esquina superior derecha, encima de la consola) y haz click en **Reload**. Refresca tu blog https://yourname.pythonanywhere.com para ver los cambios.

¡Y eso debería ser todo! Felicidades :)

¿Qué sigue?

Date muchas felicitaciones. **¡Eres increíble!** ¡Estamos orgullosas! <3

¿Qué hacer ahora?

Toma un descanso y relájate. Acabas de hacer algo realmente grande.

Después de eso, asegúrate de seguir a Django Girls en Facebook o Twitter para estar al día.

¿Me puedes recomendar recursos adicionales?

¡Claro! En primer lugar, sigue adelante y prueba nuestro otro libro llamado Django Girls Turorial: Extensions.

Luego, puedes intentar los recursos listados a continuaciòn. ¡Todos son muy recomendados!

- Django's official tutorial (Tutorial oficial de Django)
- New Coder turorials (Tutoriales de "New Coder")
- Code Academy Python course (Curso de python en Code Academy)
- Code Academy HTML & CSS course (Curso de HTML y CSS en Code Academy)
- Django Carrots tutorial (Tutorial de Django de "Carrots")
- Learn Python The Hard Way book (Libro Aprende Python a las Malas)
- Getting Started With Django video lessons (Lecciones de video Empezando con Django)
- Two Scoops of Django 1.11: Best Practices for Django Web Framework book (Libro Dos Cucharadas de Django 1.11: mejores pràcticas para Django Web Framework)
- Hello Web App: Learn How to Build a Web App (Hola aplicaciòn web: Aprende como construir una aplicaciòn web) - tambièn puedes solicitar un eBook gratis contactando con la autora Tracy Osborn en tracy@limedaring.com